



Technische  
Universität  
Braunschweig



# Correctness-by-Construction for Correct and Secure Software Systems

Von der  
Carl-Friedrich-Gauß-Fakultät  
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines  
Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation  
(kumulative Arbeit)

von  
Tobias Runge  
geboren am 22. Dezember 1993  
in Gronau (Leine)

Eingereicht am: 21.04.2023  
Disputation am: 14.04.2023  
1. Referent: Prof. Dr. Martin Johns  
2. Referentin: Prof. Dr. Ina Schaefer  
3. Referent: Prof. Dr. Reiner Hähnle



# Abstract

Ensuring the safety and security of software systems is more important today than ever before, as critical domains (such as automotive, aviation, or healthcare systems) become increasingly software-intensive. Typically, such critical software is exhaustively tested, but testing alone cannot guarantee correctness. Therefore, formal approaches are required to ensure safety and security of the developed software. For functional correctness, post-hoc verification is the state-of-the-art. By post-hoc verification, we mean that a program is implemented, specified with a pre-/postcondition contract, and then verified. However, this approach has the downside that it does not provide guidelines for developers how to write correct code. If a program developed without guidelines cannot be verified, it is often difficult to find the root causes. It may be a faulty implementation or an unsuitable specification of the program. As a result, it is costly to debug the program and to verify it again. In terms of security, it is important to ensure confidentiality and integrity of processed data. Here, static and dynamic taint analysis approaches are prevalent, where data is labeled with a security level to analyze whether there is a prohibited flow from a secure source to a public sink as defined by a security policy. This post-hoc analysis has the same disadvantage as post-hoc verification, an incorrect program must be debugged and then reanalyzed to ensure that the vulnerability is fixed. The general drawback of post-hoc analysis and verification techniques is that they reveal problems in the code, but do not assist in correcting them.

Correctness-by-Construction (CbC) is an alternative program development approach in which a program is directly constructed to be correct. Starting with a specified, but abstract program, a set of refinement rules is applied to refine the program into a concrete implementation. Each applied refinement rule guarantees that the program under development satisfies the initial specification. Thus, the resulting program is correct by construction. It is argued that with the CbC approach, defects are discovered at an early stage of development and can be resolved more easily through a structured program development approach. Despite these benefits, CbC has not yet become an established development approach for safety- and security-critical software. We identified three main reasons/challenges: (1) lack of tool support, (2) rigid program development due to the fine-grained refinement rules that allow only one program statement to be added at a time, (3) and no CbC approach to ensure non-functional properties, such as information flow security.

In this thesis, we provide remedy to the aforementioned challenges by establishing and supporting a correct-by-construction program development approach for functionally correct and secure programs. In particular, we make the following four contributions in this thesis. As a first contribution, we implement CORC, tool support for CbC. We determine the requirements to enable CbC for correct program development and develop tool support accordingly. CORC ensures that the correctness of the program under development is ensured in each construction step. In the second contribution, we conduct user studies to determine the comprehensibility and usability of CORC. In the user studies, we are able to analyze how developers proceed to solve programming tasks, what mistakes they make, and how they like using the tool. We compare these results to state-of-the-art program development with post-hoc verification. As third contribution, we address the rigid program

development approach of classic CbC by considering new approaches that allow for more flexible program construction. We introduce new refinement rules that condense the application of other refinement rules. We also propose a new trait-based CbC development approach that is based on program composition instead of refinement rules. Since CbC only considers functional correctness up to our work, we extend CbC to also ensure secure information flow in our fourth contribution. We develop refinement rules that ensure program conformance with information flow security policies. At each refinement step, the information flow is analyzed to allow only secure program statements. The refinement rules are integrated into CORC to provide tool support for the constructive development of secure software, and we demonstrate the feasibility of CORC by correctly constructing several case studies. In summary, our work enables the development of functionally correct and secure programs using correctness-by-construction. This foundation can be used to determine whether CbC is a viable alternative to post-hoc verification or post-hoc analysis.

# Zusammenfassung

Die Gewährleistung der Sicherheit (funktionale Sicherheit und Datensicherheit) von Softwaresystemen ist heute wichtiger denn je, da kritische Bereiche (wie in der Automobilindustrie, der Luftfahrt oder dem Gesundheitswesen) immer softwareintensiver werden. In der Regel wird solche kritische Software umfassend getestet, aber Tests allein können die Korrektheit nicht garantieren. Daher sind formale Ansätze erforderlich, um die Sicherheit der entwickelten Software zu gewährleisten. Für funktionale Korrektheit ist die Post-hoc-Verifikation der Stand der Technik. Unter Post-hoc-Verifikation verstehen wir, dass ein Programm implementiert, mit einem Vor-/Nachbedingungsvertrag spezifiziert und dann verifiziert wird. Dieser Ansatz hat jedoch den Nachteil, dass er den Entwicklern keinen Leitfaden für das Schreiben von korrektem Code bietet. Wenn ein ohne Leitfaden entwickeltes Programm nicht verifiziert werden kann, ist es oft schwierig, die Ursachen dafür zu finden. Es kann sich um eine fehlerhafte Implementierung oder eine ungeeignete Spezifikation des Programms handeln. Dies hat zur Folge, dass es kostspielig ist, das Programm zu debuggen und erneut zu verifizieren. Im Hinblick auf die Datensicherheit ist es wichtig, die Vertraulichkeit und Integrität der verarbeiteten Daten zu gewährleisten. Hier sind statische und dynamische Taint-Analysen üblich, bei denen die Daten mit einer Sicherheitsstufe versehen werden, um zu analysieren, ob es einen Fluss von einer sicheren Quelle zu einer öffentlichen Senke gibt, der durch eine Sicherheitsrichtlinie verboten wurde. Diese Post-hoc-Analyse hat denselben Nachteil wie die Post-hoc-Verifizierung: Ein fehlerhaftes Programm muss korrigiert und dann neu analysiert werden, um sicherzustellen, dass die Schwachstelle behoben ist. Der allgemeine Nachteil von Post-hoc-Analyse- und Verifizierungstechniken besteht darin, dass sie zwar Probleme im Code aufdecken, aber nicht bei deren Behebung helfen.

Correctness-by-Construction (CbC) ist ein alternativer Ansatz zur Programmentwicklung, bei dem ein Programm direkt so konstruiert wird, dass es korrekt ist. Ausgehend von einem spezifizierten, aber abstrakten Programm wird eine Reihe von Verfeinerungsregeln angewendet, um das Programm in eine konkrete Implementierung zu verfeinern. Jede angewandte Verfeinerungsregel garantiert, dass das zu entwickelnde Programm die ursprüngliche Spezifikation erfüllt. Somit ist das resultierende Programm konstruktionsbedingt korrekt. Es wird argumentiert, dass mit dem CbC-Ansatz Fehler in einem frühen Stadium der Entwicklung entdeckt werden und durch einen strukturierten Programmentwicklungsansatz leichter behoben werden können. Trotz dieser Vorteile hat sich CbC noch nicht als Entwicklungsansatz für sicherheitskritische Software durchgesetzt. Wir haben drei Hauptgründe beziehungsweise Herausforderungen identifiziert: (1) fehlende Werkzeugunterstützung, (2) starre Programmentwicklung aufgrund der feinkörnigen Verfeinerungsregeln, die jeweils nur eine Programmanweisung hinzufügen (3) und fehlen eines CbC-Ansatzes zur Sicherstellung nicht-funktionaler Eigenschaften, wie z.B. der Informationsflusssicherheit.

In dieser Arbeit werden die zuvor genannten Herausforderungen adressiert, indem wir einen Correct-by-Construction-Programmentwicklungsansatz für funktional korrekte und datensichere Programme etablieren und unterstützen. Insbesondere werden in dieser Arbeit die folgenden vier Beiträge geleistet. Als ersten Beitrag implementieren wir CORC, eine Werkzeugunterstützung für

CbC. Wir bestimmen die Anforderungen, um eine korrekte Programmentwicklung mit CbC zu ermöglichen, und entwickeln eine entsprechende Werkzeugunterstützung. CORC stellt sicher, dass die Korrektheit des zu entwickelnden Programms in jedem Konstruktionsschritt gewährleistet ist. Im zweiten Beitrag führen wir Benutzerstudien durch, um die Verständlichkeit und Benutzbarkeit von CORC zu ermitteln. In den Nutzerstudien können wir analysieren, wie Entwickler bei der Lösung von Programmieraufgaben vorgehen, welche Fehler sie machen und wie gerne sie das Werkzeug nutzen. Diese Ergebnisse werden mit der Programmentwicklung mit Post-hoc-Verifikation verglichen. Als dritten Beitrag adressieren wir den starren Programmentwicklungsansatz des klassischen CbC, indem wir neue Ansätze berücksichtigen, die eine flexiblere Programmkonstruktion ermöglichen. Wir führen neue Verfeinerungsregeln ein, die die Anwendung anderer Verfeinerungsregeln bündeln. Außerdem schlagen wir einen neuen Trait-basierten CbC-Entwicklungsansatz vor, der auf Programmkomposition anstelle von Verfeinerungsregeln basiert. Da CbC bis zu unserer Arbeit nur funktionale Korrektheit berücksichtigt hat, erweitern wir CbC in unserem vierten Beitrag, um auch einen sicheren Informationsfluss zu gewährleisten. Wir entwickeln Verfeinerungsregeln, die die Konformität des Programms mit den Informationsflusssicherheitsrichtlinien gewährleisten. Bei jedem Verfeinerungsschritt wird der Informationsfluss analysiert, um nur sichere Programmanweisungen zuzulassen. Die Verfeinerungsregeln werden in CORC integriert, um Werkzeugunterstützung für die konstruktive Entwicklung sicherer Software zu bieten, und wir demonstrieren die Praktikabilität von CORC, indem wir mehrere Fallstudien korrekt konstruieren. Insgesamt ermöglicht unsere Arbeit die Entwicklung von funktional korrekten und datensicheren Programmen mit Hilfe von Correctness-by-Construction. Auf dieser Grundlage kann festgestellt werden, ob CbC eine wirkliche Alternative zur Post-hoc-Verifikation oder Post-hoc-Analyse ist.

# Danksagung

Während meiner Promotion haben mich viele Menschen begleitet und unterstützt. Mit dieser Danksagung möchte ich mich bei all diesen Menschen für ihre Unterstützung, Betreuung und die vielen guten Stunden bedanken.

Zuallererst möchte ich meiner Betreuerin, Ina Schaefer, dafür danken, dass sie mir die Möglichkeit gegeben hat, in einem exzellenten Forschungs- und Lernumfeld zu promovieren, und dass sie mir sowohl in akademischer als auch in persönlicher Hinsicht eine echte Mentorin war. Beginnend mit dem Seminar am Institut (und natürlich der Projektarbeit in Stellenbosch) wurde mein Interesse an der Forschung geweckt. Ich danke auch meinem früheren Kollegen Thomas Thüm, der mich in den ersten Jahren am Institut betreut hat. Seine Unterstützung und Tipps in dieser Zeit helfen mir bis heute und haben den Weg zur Promotion geebnet. Ein großer Dank geht an Loek Cleophas und Bruce Watson, die mich immer zur Arbeit mit Correctness-by-Construction unterstützt haben. Ich danke außerdem Alex Potanin und Marco Servetto für die gelungene und lehrreiche Zusammenarbeit in den letzten Jahren meiner Promotion. Ich danke meinen Gutachtern, Prof. Reiner Hähnle und Prof. Martin Johns, dafür, dass sie sich die Zeit genommen haben, meine Dissertation zu begutachten und mir wertvolles Feedback zu geben.

Ein großer Dank geht an alle meine Kollegen vom Institut für Softwaretechnik und Fahrzeuginformatik der TU Braunschweig und der Forschungsgruppe für Test, Validierung und Analyse Software-intensiver Systeme am Karlsruher Institut für Technologie. Wir hatten eine schöne und besondere Zeit mit Kickern, Weihnachtsfeiern, Betriebsausflügen, Braunkohlwanderungen und den Mittagspausen, aber auch produktive Treffen im Formal Methods Club mit Tabea Bordis und Alexander Kittelmann. Mit euch war es immer eine Freude, wissenschaftlich zu arbeiten oder die wirklich spannende Zeit zu genießen. Einen besonderen Platz haben die zahlreichen Dienstreisen mit Kollegen in Porto, Göteborg, Berlin, Amsterdam, .... All diese Trips bleiben unvergesslich in Erinnerung. Außerdem danke ich allen Kollegen aus Stellenbosch, die mich während meiner Zeit dort freundlich und hilfsbereit aufgenommen haben.

Schließlich danke ich meiner Familie für ihre enorme Unterstützung. Meine Mutter Martina und mein Bruder Tjard sind immer für mich da. Außerdem danke ich allen Freunden, die meine gute Laune oben gehalten haben. Ja, ihr alle seid gemeint: aus der Heimat, vom Abi, aus der Pumpe und die Gilde Drinks per Second. Ich kann euch hier nicht alle namentlich erwähnen, aber ihr wisst, dass ihr mir sehr viel bedeutet.

Ob ihr mich nun wissenschaftlich oder menschlich unterstützt habt oder in allen Lebenslagen da wart, ohne euch alle wäre diese Arbeit niemals zustande gekommen.





# Publications

The publications that are part of this cumulative thesis are listed below.

- [1] **Runge, T.**, I. Schaefer, L. Cleophas, T. Thüm, D. Kourie, and B. W. Watson (2019a). “Tool Support for Correctness-by-Construction”. In: *International Conference on Fundamental Approaches to Software Engineering*. Vol. 11424. Lecture Notes in Computer Science. Springer, pp. 25–42. DOI: [10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2).
- [2] **Runge, T.**, T. Thüm, L. Cleophas, I. Schaefer, and B. W. Watson (2019b). “Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study”. In: *Formal Methods. FM 2019 International Workshops. Refine*. Vol. 12233. Lecture Notes in Computer Science. Springer, pp. 388–405. DOI: [10.1007/978-3-030-54997-8\\_25](https://doi.org/10.1007/978-3-030-54997-8_25).
- [3] **Runge, T.**, T. Bordis, T. Thüm, and I. Schaefer (2021a). “Teaching Correctness-by-Construction and Post-hoc Verification—The Online Experience”. In: *Formal Methods Teaching Workshop*. Vol. 13122. Lecture Notes in Computer Science. Springer, pp. 101–116. DOI: [10.1007/978-3-030-91550-6\\_8](https://doi.org/10.1007/978-3-030-91550-6_8).
- [4] **Runge, T.**, A. Potanin, T. Thüm, and I. Schaefer (2022a). “Traits: Correctness-by-Construction for Free”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Vol. 13273. Lecture Notes in Computer Science. Springer, pp. 131–150. DOI: [10.1007/978-3-031-08679-3\\_9](https://doi.org/10.1007/978-3-031-08679-3_9).
- [5] **Runge, T.**, T. Bordis, A. Potanin, T. Thüm, and I. Schaefer (2023). “Flexible Correct-by-Construction Programming”. *Logical Methods in Computer Science*.
- [6] **Runge, T.**, M. Servetto, A. Potanin, and I. Schaefer (2022b). “Immutability and Encapsulation for Sound OO Information Flow Control”. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. DOI: [10.1145/3573270](https://doi.org/10.1145/3573270).
- [7] **Runge, T.**, A. Knüppel, T. Thüm, and I. Schaefer (2020). “Lattice-Based Information Flow Control-by-Construction for Security-by-Design”. In: *FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering*. ACM, pp. 44–54. DOI: [10.1145/3372020.3391565](https://doi.org/10.1145/3372020.3391565).
- [8] **Runge, T.**, A. Kittelmann, M. Servetto, A. Potanin, and I. Schaefer (2022c). “Information Flow Control-by-Construction for an Object-Oriented Language”. In: *International Conference on Software Engineering and Formal Methods*. Vol. 13550. Lecture Notes in Computer Science. Springer, pp. 209–226. DOI: [10.1007/978-3-031-17108-6\\_13](https://doi.org/10.1007/978-3-031-17108-6_13).

### Further peer-reviewed publications that are not part of this cumulative thesis.

- [9] Schaefer, I., **T. Runge**, A. Knüppel, L. Cleophas, D. Kourie, and B. W. Watson (2018). “Towards Confidentiality-by-Construction”. In: *International Symposium on Leveraging Applications of Formal Methods*. Vol. 11244. Lecture Notes in Computer Science. Springer, pp. 502–515. DOI: [10.1007/978-3-030-03418-4\\_30](https://doi.org/10.1007/978-3-030-03418-4_30).
- [10] **Runge, T.**, I. Schaefer, A. Knüppel, L. Cleophas, D. Kourie, and B. W. Watson (2019c). “Tool Support for Confidentiality-by-Construction”. *ACM SIGAda Ada Letters* 38.2, pp. 64–68. DOI: [10.1145/3375408.3375413](https://doi.org/10.1145/3375408.3375413).
- [11] Knüppel, A., **T. Runge**, and I. Schaefer (2020a). “Scaling Correctness-by-Construction”. In: *International Symposium on Leveraging Applications of Formal Methods*. Ed. by T. Margaria and B. Steffen. Vol. 12476. Lecture Notes in Computer Science. Springer, pp. 187–207. DOI: [10.1007/978-3-030-61362-4\\_10](https://doi.org/10.1007/978-3-030-61362-4_10).
- [12] Bordis, T., L. Cleophas, A. Kittelmann, **T. Runge**, I. Schaefer, and B. W. Watson (2022a). “Re-CorC-ing KeY: Correct-by-Construction Software Development Based on KeY”. In: *The Logic of Software. A Tasting Menu of Formal Methods*. Vol. 13360. Lecture Notes in Computer Science. Springer. DOI: [10.1007/978-3-031-08166-8\\_5](https://doi.org/10.1007/978-3-031-08166-8_5).
- [13] Bordis, T., M. Kodetzki, **T. Runge**, and I. Schaefer (2022b). “VarCorC: Developing Object-Oriented Software Product Lines Using Correctness-by-Construction”. In: *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops F-IDE*. Vol. 13765. Lecture Notes in Computer Science. Springer, pp. 156–163. DOI: [10.1007/978-3-031-26236-4\\_13](https://doi.org/10.1007/978-3-031-26236-4_13).
- [14] Bordis, T., **T. Runge**, and I. Schaefer (2020a). “Correctness-by-Construction for Feature-Oriented Software Product Lines”. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. ACM, pp. 22–34. DOI: [10.1145/3425898.3426959](https://doi.org/10.1145/3425898.3426959).
- [15] Bordis, T., **T. Runge**, A. Knüppel, T. Thüm, and I. Schaefer (2020b). “Variational Correctness-by-Construction”. In: *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS)*. ACM, 7:1–7:9. DOI: [10.1145/3377024.3377038](https://doi.org/10.1145/3377024.3377038).
- [16] Bordis, T., **T. Runge**, D. Schultz, and I. Schaefer (2022c). “Family-Based and Product-Based Development of Correct-by-Construction Software Product Lines”. *Journal of Computer Languages*, p. 101119. DOI: [10.1016/j.cola.2022.101119](https://doi.org/10.1016/j.cola.2022.101119).
- [17] Bordis, T., **T. Runge**, A. Kittelmann, and I. Schaefer (2022d). “Correctness-by-Construction: An Overview of the CorC Ecosystem”. *ACM SIGAda Ada Letters*.
- [18] Knüppel, A., I. Jatzkowski, M. Nolte, T. Thüm, **T. Runge**, and I. Schaefer (2020b). “Skill-Based Verification of Cyber-Physical Systems”. In: *International Conference on Fundamental Approaches to Software Engineering*. Ed. by H. Wehrheim and J. Cabot. Vol. 12076. Lecture Notes in Computer Science. Springer, pp. 203–223. DOI: [10.1007/978-3-030-45234-6\\_10](https://doi.org/10.1007/978-3-030-45234-6_10).
- [19] Kittelmann, A., **T. Runge**, T. Bordis, and I. Schaefer (2022). “Runtime Verification of Correct-by-Construction Driving Maneuvers”. In: *International Symposium on Leveraging Applications of*

*Formal Methods*. Vol. 13701. Lecture Notes in Computer Science. Springer, pp. 242–263. DOI: [10.1007/978-3-031-19849-6\\_15](https://doi.org/10.1007/978-3-031-19849-6_15).

- [20] Kuitert, E., A. Knüppel, T. Bordis, **T. Runge**, and I. Schaefer (2022). “Verification Strategies for Feature-Oriented Software Product Lines”. In: *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. ACM, 12:1–12:9. DOI: [10.1145/3510466.3511272](https://doi.org/10.1145/3510466.3511272).
- [21] Pett, T., S. Krieter, **T. Runge**, T. Thüm, M. Lochau, and I. Schaefer (2021). “Stability of Product-Line Sampling in Continuous Integration”. In: *VaMoS’21: 15th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, 18:1–18:9. DOI: [10.1145/3442391.3442410](https://doi.org/10.1145/3442391.3442410).
- [22] Pett, T., T. Thüm, **T. Runge**, S. Krieter, M. Lochau, and I. Schaefer (2019). “Product Sampling for Product Lines: The Scalability Challenge”. In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM. DOI: [10.1145/3336294.3336322](https://doi.org/10.1145/3336294.3336322).
- [23] **Runge, T.**, I. Schaefer, L. Cleophas, and B. W. Watson (2017). “Many-MADFAct: Concurrently Constructing MADFAs”. In: *Proceedings of the Prague Stringology Conference*, pp. 126–142.
- [24] Varshosaz, M., M. Al-Hajjaji, T. Thüm, **T. Runge**, M. R. Mousavi, and I. Schaefer (2018). “A Classification of Product Sampling for Software Product Lines”. In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, pp. 1–13. DOI: [10.1145/3233027.3233035](https://doi.org/10.1145/3233027.3233035).
- [25] Wille, D., **T. Runge**, C. Seidl, and S. Schulze (2017). “Extractive Software Product Line Engineering Using Model-based Delta Module Generation”. In: *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. ACM, pp. 36–43. DOI: [10.1145/3023956.3023957](https://doi.org/10.1145/3023956.3023957).
- [26] Becker, M., R. Meyer, **T. Runge**, I. Schaefer, S. van der Wall, and S. Wolff (2022). “Model-Based Fault Classification for Automotive Software”. *arXiv preprint arXiv:2208.14290*. DOI: [10.1007/978-3-031-21037-2\\_6](https://doi.org/10.1007/978-3-031-21037-2_6).
- [27] Knüppel, A., I. Jatzkowski, M. Nolte, **T. Runge**, T. Thüm, and I. Schaefer (2021). “Skill-Based Verification of Cyber-Physical Systems”. In: *Software Engineering 2021, Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26. Februar 2021, Braunschweig/Virtuell*. Ed. by A. Koziolok, I. Schaefer, and C. Seidl. Vol. P-310. LNI. Gesellschaft für Informatik e.V., pp. 67–68. DOI: [10.18420/SE2021\\_22](https://doi.org/10.18420/SE2021_22).
- [28] **Runge, T.**, I. Schaefer, L. Cleophas, T. Thüm, D. G. Kourie, and B. W. Watson (2021b). “Tool Support for Correctness-by-Construction”. In: *Software Engineering 2021, Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26. Februar 2021, Braunschweig/Virtuell*. Ed. by A. Koziolok, I. Schaefer, and C. Seidl. Vol. P-310. LNI. Gesellschaft für Informatik e.V., pp. 93–94. DOI: [10.18420/SE2021\\_34](https://doi.org/10.18420/SE2021_34).

- [29] Schaefer, I., **T. Runge**, L. Cleophas, and B. W. Watson (2021). “Tutorial: The Correctness-by-Construction Approach to Programming Using CorC”. In: *IEEE Secure Development Conference, SecDev 2021*. IEEE, pp. 1–2. DOI: [10.1109/SecDev51306.2021.00012](https://doi.org/10.1109/SecDev51306.2021.00012).

# Contents

<b>List of Figures</b>	<b>iii</b>
<b>List of Tables</b>	<b>v</b>
<b>List of Code Listings</b>	<b>vii</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Challenges for Correct-by-Construction Program Development . . . . .	2
1.2. Research Questions . . . . .	5
1.3. Contributions . . . . .	6
1.4. Reader’s Guide . . . . .	9
<b>2. Background</b>	<b>11</b>
2.1. Contracts and Contract-based Verification . . . . .	11
2.1.1. Method Contracts . . . . .	11
2.1.2. Program Verification . . . . .	13
2.2. Correctness-by-Construction . . . . .	14
2.3. Information Flow Control . . . . .	18
<b>3. Contributions</b>	<b>21</b>
3.1. CoRC Tool Support for Functional Correct-by-Construction Program Development .	21
3.2. A Usability Evaluation of the CbC Approach as Implemented in CoRC . . . . .	29
3.3. Alternative Correct-by-Construction Program Development Approaches . . . . .	35
3.4. A Uniform Correct-by-Construction Program Development Approach for Functional Correctness and Security . . . . .	44
<b>4. Conclusion</b>	<b>53</b>
4.1. Discussion of Research Questions . . . . .	53
4.2. Discussion of the Main Research Question . . . . .	55
4.3. Ongoing and Future Work . . . . .	56
<b>Bibliography</b>	<b>59</b>
<b>A. Papers of the Thesis</b>	<b>71</b>
A.1. Tool Support for Correctness-by-Construction . . . . .	72
A.2. Comparing Correctness-by-Construction with Post-Hoc Verification — A Qualita- tive User Study . . . . .	91
A.3. Teaching Correctness-by-Construction and Post-hoc Verification — The Online Ex- perience . . . . .	110
A.4. Traits: Correctness-by-Construction for Free . . . . .	127

A.5. Flexible Correct-by-Construction Programming . . . . .	148
A.6. Immutability and Encapsulation for Sound OO Information Flow Control . . . . .	185
A.7. Lattice-Based Information Flow Control-by-Construction for Security-by-Design . . . . .	222
A.8. Information Flow Control-by-Construction for an Object-Oriented Language . . . . .	234

# List of Figures

2.1. Refinement steps for the linear search algorithm . . . . .	17
2.2. Security type system [Volpano et al. 1996] . . . . .	20
3.1. Linear search algorithm in the graphical IDE . . . . .	23





# List of Tables

3.2. Defects in code and specification of the final programs of participants [Runge et al. 2021]	31
3.4. Comparison of classic CbC with CBC-BLOCK and TRAITCbC [Runge et al. 2023] . . . .	42



# List of Code Listings

2.1. An implementation of a linear search algorithm specified with JML . . . . .	12
2.2. Examples of of well-typed and ill-typed statements . . . . .	19
3.1. Initial program of <code>maxElement</code> . . . . .	37
3.2. Refinement of block B1 . . . . .	38
3.3. Initial trait for <code>maxElement</code> . . . . .	39
3.4. Implementation of <code>maxElement</code> with auxiliary methods . . . . .	40
3.5. Class declarations . . . . .	46
3.6. Examples with immutable objects . . . . .	47
3.7. Examples with mutable and encapsulated objects . . . . .	47



# 1. Introduction

The amount of software in safety-critical and security-critical systems increases, and therefore, it gets more difficult to ensure functional correctness and security of these software systems. In areas, such as the automotive industry, aviation, or healthcare systems, incorrect software can cost not only money, but also human lives. The correctness of software in the mentioned areas is usually checked with testing, but tests do not provide high safety guarantees. A formal approach is necessary to guarantee correctness, as is already required in aviation for high design assurance levels [RTCA DO-178C 2011]. In this thesis, we focus on deductive verification of specified programs to guarantee functional correctness.

In this context, post-hoc verification is state-of-the-art, where a program is verified after it is fully implemented [Ahrendt et al. 2016]. The predominant approach for correctly designing software is *Design-by-Contract*, which is used to ensure the correctness of the software at method level [Meyer 1992; Meyer 1988]. With design-by-contract, methods are specified by pre-/postcondition contracts. A method is correct if the method fulfills its contract (i.e., if the program is in a state that fulfills the precondition, the method terminates, and then the program is in a state that fulfills the postcondition). To construct software, developers<sup>1</sup> may start with a formal specification of the problem, but they are not given guidelines on *how* to construct the program (e.g., specific construction rules to create the method). The resulting implementation is then verified against the starting specification. For example, post-hoc verification tools such as KEY [Ahrendt et al. 2016] or languages with integrated verification tools such as Dafny [Leino 2010] (semi-)automatically verify software with respect to its formal specification. Once the software, which was developed without any guidelines, cannot be verified, it is difficult to identify the source of the defects. In particular, non-conformance between specification and implementation can be due to three reasons: an incorrect method implementation, an incorrect or unsuitable formal specification, or the verification tool was unable to find a proof automatically (i.e., due to exceeded resource limits or general undecidability). However, finding the exact parts (or even statements) of the program that can be attributed to the non-conformance is oftentimes impossible. Furthermore, the developer's expertise certainly plays a key role in inspecting the root causes for a failed verification [Knüppel et al. 2018]. For example, beginners often have difficulty verifying any meaningful algorithm beyond toy examples. The more complex a method is, the more challenging it is to verify its correctness.

An alternative approach to design-by-contract is the *direct construction* of correct programs, such as in the incremental refinement- and rule-based correctness-by-construction (CbC) approach [Dijkstra 1976; Kourie and Watson 2012; Morgan 1994; Gries 1987; Wirth 1971]. With CbC, a specified program is incrementally refined to concrete code using a set of refinement rules that preserve the correctness of the program with respect to the specification. The correctness of the whole program is guaranteed through side conditions that are defined in the refinement rules. In each refinement step, an abstract statement (i.e., a hole in the program) is refined to a more concrete implemen-

---

<sup>1</sup>In this thesis, we are using the term developer for someone who specifies, implements, and verifies software. In practice, this could be three different roles.

tation that can still contain some nested abstract statements. The construction ends when no abstract statement is left. The main idea of this specification-first, refinement-based approach is that developers are forced to think about their implementation more thoroughly rather than having a trial-and-error verification approach. With the structured reasoning discipline that is enforced by the refinement rules in CbC, defects are more likely to be discovered at an early stage and can be tracked through the program. Thus, it is argued that program quality is increased and verification effort is reduced [Kourie and Watson 2012; Watson et al. 2016].

Beside functional correctness, useful software has to fulfill further non-functional requirements [ISO/IEC 25010 2011], such as timing behavior, resource consumption, and capacity consumption. In this thesis, we focus on security properties because it is a major challenge to protect user data and ensure the integrity of systems against attackers. To express confidentiality and integrity of information, we consider information flow policies. An information flow policy specifies the allowed and prohibited flow of information between different security levels. Static and dynamic analysis techniques [Sabelfeld and Myers 2003; Russo and Sabelfeld 2010] or Hoare-style program logics [Darvas et al. 2005; Amtoft et al. 2006; Beckert et al. 2013] are used to ensure information flow policies. With dynamic [Enck et al. 2014; Hedin et al. 2014] or static [Arzt et al. 2014; Huang et al. 2014; Graf et al. 2013] taint analysis, the flow of data sources to sinks in a program can be analyzed after the program is fully developed. If there is a flow from a private source to a public sink, the information flow policy is violated, and an information leak is detected. To close this leak, the program must be debugged, and then checked again whether the information flow leak is now fixed. The difficulty is that the analysis techniques indicate the leak, but do not help to fix the problem. Type systems [Sabelfeld and Myers 2003] are useful to ensure correct information flows more directly during the construction of programs. With type systems, variables are labeled with security levels, and leaks are detected if the type system cannot type-check the program. But even developers, who use a security type system, can face serious problems during programming if they do not design the security properties from the beginning. Starting with a security specification and having an incremental guideline how to construct a program helps to determine the allowed flows in each construction step and simplifies the debugging of security violations.

The post-hoc verification approach for ensuring functional correctness [Ahrendt et al. 2016; Leino 2010] and the techniques for ensuring secure information flow [Sabelfeld and Myers 2003; Graf et al. 2013; Amtoft et al. 2006] are state-of-the-art. As mentioned, these approaches can lead to tedious verification and debugging efforts if the software contains defects. In this thesis, we investigate to what extent it is possible to construct correct and secure programs with CbC.

## 1.1. Challenges for Correct-by-Construction Program Development

Since CbC has not yet caught on as an alternative program development approach, we investigate the challenges to enable and support a CbC approach for functionally correct and secure programs. These challenges that have to be addressed are presented in the following.

For CbC [Dijkstra 1976; Gries 1987], there exist some approaches and tools to support developers during program construction, such as ArcAngel [Oliveira et al. 2003], SOCOS [Back 2009;

Back et al. 2007], or Dafny [Leino 2010; Ettinger 2021]. The language ArcAngel implemented with ProofPower [Zeyda et al. 2009] supports Morgan’s refinement calculus [Morgan 1994]. The calculus offers a wide variety of refinement rules to construct programs. These refinement rules are bundled in tactics, which when applied, incrementally transform the starting specification to a concrete implementation satisfying this specification. Each applied tactic discharges proof obligations that must be proven for the refinement to be correct. SOCOS [Back 2009; Back et al. 2007] uses a graphical interface where programs are constructed as UML-style state chart. Starting with an invariant specification, a program is incrementally refined to fulfill the invariant. SOCOS is used to teach refinement-based programming [Back 2009]. In user studies, Back [2009] determined that good tool support is necessary for refinement-based program development because otherwise the application of refinements is overwhelming for developers. Furthermore, Back [2009] detected that finding invariants is the biggest issue during program construction. Developers use most of the time to refine an incomplete or partially incorrect invariant to a correct solution [Back 2009]. Quite differently, Dafny [Leino 2010] is a verification-aware programming language which integrates specification writing into the programming workflow. A verifier constantly proves whether the source code satisfies the specification to show errors or to confirm correctness. With Dafny, developers are free to implement correct software without using a specific process, or they can use small-step refinement processes as described by Ettinger [2021]. However, for these small-step refinement processes no specialized tool support is available.

None of these CbC approaches has become accepted as an alternative to post-hoc verification.<sup>2</sup> We argue that one reason is the applicability of the CbC approach for developers. We assume, the previous mentioned approaches are too complex to use, or specialized tool support is not available as in the case of Dafny. Developers must have a real benefit over post-hoc approaches (e.g. simplicity of the approach, or reduced specification and verification effort during program development) to engage in a correct-by-construction program development approach. CbC introduced by Kourie and Watson [2012] is centered around comprehensibility and usability for developers. They reduced the set of refinement rules to a small, but useful set without losing expressiveness, so that developers do not have to deal with refinement rules that only formal method experts are interested in [Kourie and Watson 2012]. Therefore, this CbC approach has the potential to be applicable and comprehensible if adequate tool support is developed, and the approach is still expressive enough to implement sufficiently complex case studies. Since there is no tool support yet, an open challenge is to implement tool support, so that, in a next step, it can be investigated whether CbC by Kourie and Watson [2012] is suitable for correct program development as an alternative to post-hoc verification.

#### Challenge 1: Tool Support for Functional Correctness-by-Construction

The correct-by-construction program development approach by Kourie and Watson [2012] is the best candidate to be usable for software developers without a major background in formal methods, but tool support is missing. To facilitate the advantages of CbC in general and to enable the evaluation of CbC by Kourie and Watson [2012] against post-hoc verification in particular, the challenge is to provide tool support and evaluate its suitability.

<sup>2</sup>We will just write post-hoc verification in this thesis, but we mean the whole process of specifying and constructing programs, and verifying them afterwards.

The CbC refinement rules by Kourie and Watson [2012] give a strict guideline how to construct programs. The drawback is that the flexibility of creating a program is limited to the existing set of refinement rules and the rigid, rule-based construction approach. It is not possible to deviate from this rule-based program construction approach. Furthermore, the refinement rules given by Kourie and Watson [2012] are relatively fine-grained by introducing only one language construct (e.g., branching or assignment statement) at a time. This leads to a lot of overhead in the construction of programs. For example, it is tedious to introduce consecutive assignment statements, even if these statements are obviously correct according to the specification. As a result, the concepts of CbC seems tedious and demand high effort and necessary knowledge from the developer to construct programs [Back 2009; Runge et al. 2021]. We therefore formulate the second challenge addressed in this thesis as follows.

#### Challenge 2: Restricted Flexibility of Rule-based Correctness-by-Construction

One drawback of rule-based CbC that limits its usability and flexibility are the fine-grained refinement rules. The CbC approach requires new or alternative concepts to relax rigid program development, but which still guarantee correctness at each development step.

The third challenge focuses on information flow security to guarantee confidentiality and integrity of data. We decided to ensure this non-functional property because information flow security is a fundamental requirement of applications to prohibit unauthorized reading and manipulation of data. If functional correctness and secure information flow of a program are considered as two independent properties and ensured one after the other during program development, problems can occur. For example, the functional correctness of a program is already verified with one tool. If an information flow analysis with another tool detects a security issue, the developer must debug the program to ensure secure information flow, but then the program must be reverified because the changes could affect functional correctness. Thus, an extensive process of debugging, verifying functional correctness, and analyzing information flow is required until the program is functionally correct and secure. This problem can be mitigated by ensuring both properties simultaneously during program development. If only one process and tool is used, context switches are not necessary which can be error prone. Furthermore, when functional correctness and security are guaranteed by construction in each development step, the need for additional post-hoc verification or information flow analysis is reduced or even superfluous. We already presented that CbC has advantages for the development of functionally correct programs [Watson et al. 2016]. The characteristics of an incremental and guided program construction approach can also facilitate the development of secure programs. However, there is no CbC approach to incrementally develop programs that are functionally correct and secure.

#### Challenge 3: Correctness-by-Construction for Information Flow Security

It is tedious and error-prone to develop programs that are functionally correct and ensure secure information flow using various processes and tools. A CbC approach for information flow security is needed for the purpose of a uniform correct-by-construction program development approach for correct and secure programs.



## 1.2. Research Questions

On the basis of the identified challenges, we formulate the main research question that we want to answer in this thesis:

### Main Research Question

How can we enable and support a correct-by-construction program development approach for functionally correct and secure software?

We divide this main research question into four more specific sub-research questions to focus on relevant aspects of the overall research goal to support correct-by-construction program development.

**Research Question RQ<sub>1</sub> – Tool Support for Functional Correctness-by-Construction.** *How can we support a correct-by-construction program development approach for functionally correct software?* In **Challenge 1**, we identified that correctness-by-construction, as proposed by Kourie and Watson [2012], has the potential to be usable by software developers without a major background in formal methods, but no tool is yet provided. Within this research question, we aim to investigate the necessary criteria for translating the theoretical concepts of CbC into appropriate tool support. In particular, we want to determine the concrete requirements to support the defined concepts of CbC, and we want to find out how to implement adequate tool support based on these requirements. Since it is important to be usable in practice, our tool, to be called CORC, should be capable to support the refinement-based construction of sufficiently complex case studies. Addressing this research question is the basis for all further research questions.

**Research Question RQ<sub>2</sub> – Usability of Tool-Supported Correctness-by-Construction.** *How usable is the correct-by-construction program development approach with CORC?* With this research question, we want to investigate how to assess the comprehensibility and usability of CbC as implemented in CORC. A usability evaluation can confirm the claimed benefits of a structured development approach in practice, as well as potentials to improve CORC. Since post-hoc program verification is state-of-the-art, we want to evaluate how CORC compares to tool-supported post-hoc verification. We investigate how to conduct qualitative user studies where we can analyze how participants interact with CbC and post-hoc verification tools to implement correct programs.

**Research Question RQ<sub>3</sub> – Alternatives to Rule-based Correctness-by-Construction.** *What alternative correct-by-construction program development approaches exist, and how do they compare to rule-based correctness-by-construction?* In **Challenge 2**, we explained that rule-based CbC has a drawback due to the inflexible refinement rules. We argue that more flexible program development approaches with relaxed guidelines are beneficial to effectively construct correct programs, and we aim to evaluate this hypothesis qualitatively. In particular, we want to find alternative CbC approaches within this research question and compare them with the correct-by-construction program development approach by Kourie and Watson [2012] to assess the benefits of more flexible program development approaches.

**Research Question RQ4 – Correctness-by-Construction for Information Flow Security.** *How can we support the development of programs with correctness-by-construction that are functionally correct and satisfy information flow security?* We argue that CbC is not only limited to functional properties, but can be generalized to non-functional properties as well — including its advantages. With the fourth research question, we investigate to what extent it is possible to establish a CbC program development approach for functionally correct programs that also satisfy security properties. Considering **Challenge 3**, we investigate how to incorporate information flow analysis into a refinement-based CbC approach. To fully support a uniform correct-by-construction program development approach, we investigate how to integrate CbC for the information flow properties into the CORC tool.

### 1.3. Contributions

In this thesis, we enable and support a correct-by-construction program development approach for functionally correct and secure programs. We made the following contributions to answer the research questions.

**Contribution 1: CORC Tool Support for Functional Correct-by-Construction Program Development.**

In this contribution, we propose tool support for correctness-by-construction by Kourie and Watson [2012], called CORC [Runge et al. 2019a]. CORC supports the incremental refinement-based approach of CbC that guarantees the functional correctness of programs in each refinement step. We offer a textual and a graphical editor with their own characteristics that are described in this work. The graphical view visualizes the refinement steps explicitly. This is helpful to track problems during program construction. The textual view is designed to provide a familiar development environment for writing code enhanced with refinement keywords. For each applied refinement rule, CORC directly checks that the side conditions for applicability are satisfied. The side conditions are translated to proof obligations that are discharged by the program verifier KEY. We implement case studies to evaluate the feasibility of CORC. We also compare the verification effort of CORC in comparison to post-hoc verification, where a specified method is directly verified with KEY. CORC was presented at the International Conference on Fundamental Approaches to Software Engineering in 2019. The full paper is attached in [Section A.1](#).

T. Runge, I. Schaefer, L. Cleophas, T. Thüm, D. Kourie, and B. W. Watson [2019a]. “Tool Support for Correctness-by-Construction”. In: *International Conference on Fundamental Approaches to Software Engineering*. Vol. 11424. Lecture Notes in Computer Science. Springer, pp. 25–42. DOI: [10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2)

**Contribution 2: A Usability Evaluation of the CbC Approach as Implemented in CORC.** In this contribution, we evaluate the usability of CbC implemented in CORC by conducting two user studies [Runge et al. 2019b; Runge et al. 2021]. We compare correct-by-construction program development with post-hoc verification to get insights whether CbC is comprehensible and usable for software developers. The participants of the user studies have to implement algorithms with both approaches. We analyze their resulting implementations and intermediate snapshots of their imple-

mentations in terms of defects. By examining the intermediate snapshots, we can analyze the applied programming procedure of the participants. We get insights, how they implement the algorithms, how they write auxiliary specification (e.g., loop invariants), and when they try to verify the correctness of the implementation. Furthermore, we collect feedback of the participants with a user experience questionnaire and a structured interview. In the user experience questionnaire, the participants are asked to rate the used tools by selecting appropriate adjectives that describe the tool. In the interview, we ask the participants open questions about their preferred development procedure. The first user study was presented at the Refinement Workshop in 2019, and the second user study was presented at the Formal Methods Teaching Workshop in 2021. The full papers are attached in [Section A.2](#) and [Section A.3](#).

T. Runge, T. Thüm, L. Cleophas, I. Schaefer, and B. W. Watson [2019b]. “Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study”. In: *Formal Methods. FM 2019 International Workshops. Refine*. Vol. 12233. Lecture Notes in Computer Science. Springer, pp. 388–405. DOI: [10.1007/978-3-030-54997-8\\_25](https://doi.org/10.1007/978-3-030-54997-8_25)

T. Runge, T. Bordis, T. Thüm, and I. Schaefer [2021]. “Teaching Correctness-by-Construction and Post-hoc Verification—The Online Experience”. In: *Formal Methods Teaching Workshop*. Vol. 13122. Lecture Notes in Computer Science. Springer, pp. 101–116. DOI: [10.1007/978-3-030-91550-6\\_8](https://doi.org/10.1007/978-3-030-91550-6_8)

**Contribution 3: Alternative Correct-by-Construction Program Development Approaches.** In this contribution, we present alternative correctness-by-construction approaches which we compare with rule-based correctness-by-construction as proposed by Kourie and Watson [2012] (classic CbC). In particular, we propose CBC-BLOCK and TRAITCBC with the goal of addressing the inflexibility of classic CbC to incrementally construct correct programs. We qualitatively discuss the characteristics of the three CbC approaches (classic CbC, CBC-BLOCK, and TRAITCBC).

CBC-BLOCK, the first CbC development approach we present, aims to relax the strict guideline of classic CbC. CBC-BLOCK introduces refinement rules that condense the application of several refinement rules to improve flexibility. A specified statement can be refined to any block of code that fulfills the specification. Thus, these new refinement rules increase the ways in which programs can be developed. The idea of these refinement rules is similar to a method call, but a block can have side effects on any local variables in the program under construction. We implement CBC-BLOCK as an extension of CORC and evaluate the usability.

TRAITCBC, the second alternative development approach, relies on traits as a flexible and reusable language construct. TRAITCBC is composition-based. This means, small units of code (methods) are constructed in isolation with any approach and verified (e.g., post-hoc verification, CbC by Kourie and Watson [2012], ...). Afterwards, these methods are composed to larger programs correct by construction. The composition-based CbC program development approach ensures that the composition of correct units produces a correct result. The advantage of TRAITCBC is that the composition does not need refinement rules. We present the foundations of TRAITCBC, and implement TRAITCBC on the basis of Java with KEY as verification tool.

TRAITCBC was presented at the International Conference on Formal Techniques for Distributed Objects, Components, and Systems. Our work was awarded the best paper award. As an extended publication, we introduced CBC-BLOCK in the special issue of the Journal on Logical Methods in Computer Science. The full papers are attached in [Section A.4](#) and [Section A.5](#).

T. Runge, A. Potanin, T. Thüm, and I. Schaefer [2022b]. “Traits: Correctness-by-Construction for Free”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Vol. 13273. Lecture Notes in Computer Science. Springer, pp. 131–150. DOI: [10.1007/978-3-031-08679-3\\_9](https://doi.org/10.1007/978-3-031-08679-3_9)

T. Runge, T. Bordis, A. Potanin, T. Thüm, and I. Schaefer [2023]. “Flexible Correct-by-Construction Programming”. *Logical Methods in Computer Science*

**Contribution 4: A Uniform Correct-by-Construction Program Development Approach for Functional Correctness and Security.**

In this contribution, we introduce correctness-by-construction for information flow security. First, we create a new information flow security type system for an object-oriented language, called SIFO [Runge et al. 2022c]. This type system checks whether developed programs fulfill the defined information flow policy. To ensure correctness of the type system, we prove that private data never influences public data. This noninterference property is a central criterion for secure information flow. An attacker should never deduce private data by observing data with lower security levels. The type system is implemented and evaluated with several case studies to demonstrate feasibility of the approach. We also evaluate expressiveness of the type system in comparison to other information flow analysis tools by implementing and checking methods of a benchmark.

To establish a uniform correct-by-construction program development approach, we transform the typing rules of SIFO into refinement rules [Runge et al. 2022a]. In a previous work, we established refinement rules for a secure imperative language [Runge et al. 2020]. The refinement rules ensure that each introduced language construct fulfills the defined information flow policy [Runge et al. 2022a]. Therefore, we can guarantee that the program is secure in each refinement step. We prove soundness of the transformation by showing that each constructed program is also typable in SIFO. By integrating the refinement rules for security in CoRC, we support CbC for functional correctness and information flow security. The feasibility of CoRC is evaluated with the same case studies that are used to evaluate SIFO.

SIFO was presented in the Journal ACM Transactions on Programming Languages and Systems (TOPLAS). CbC for secure information flow in an imperative language was presented at the International Conference of Formal Methods in Software Engineering. The CbC approach for an object-oriented language that is based on SIFO was presented at the International Conference on Software Engineering and Formal Methods. The full paper are attached in [Section A.6](#), [Section A.7](#), and [Section A.8](#).

T. Runge, M. Servetto, A. Potanin, and I. Schaefer [2022c]. “Immutability and Encapsulation for Sound OO Information Flow Control”. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. DOI: [10.1145/3573270](https://doi.org/10.1145/3573270)

T. Runge, A. Knüppel, T. Thüm, and I. Schaefer [2020]. “Lattice-Based Information Flow Control-by-Construction for Security-by-Design”. In: *FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering*. ACM, pp. 44–54. DOI: [10.1145/3372020.3391565](https://doi.org/10.1145/3372020.3391565)

T. Runge, A. Kittelmann, M. Servetto, A. Potanin, and I. Schaefer [2022a]. “Information Flow Control-by-Construction for an Object-Oriented Language”. In: *International Conference on Software Engineering and Formal Methods*. Vol. 13550. Lecture Notes in Computer Science. Springer, pp. 209–226. DOI: [10.1007/978-3-031-17108-6\\_13](https://doi.org/10.1007/978-3-031-17108-6_13)

## 1.4. Reader's Guide

The thesis is divided into four chapters. In [Chapter 2](#), we provide foundations of software verification and rule-based correctness-by-construction as background for all following chapters.

In [Chapter 3](#), the main contributions are presented. The contributions of the thesis are grouped into four parts and answer the four research questions **RQ1** – **RQ4**.

In [Section 3.1](#), we present CoRC that has a textual and a graphical editor to construct programs with CbC. We also evaluate the feasibility of CoRC by implementing several case studies.

In [Section 3.2](#), we evaluate the usability of CoRC by conducting two user studies. We compare correct-by-construction software development with post-hoc verification to get insights whether CbC is comprehensible and usable for software developers.

In [Section 3.3](#), we propose alternative correctness-by-construction approaches, called TRAITCBC and CBC-BLOCK. We then compare these alternative CbC approaches with classic CbC regarding their flexibility to construct programs.

In [Section 3.4](#), we introduce SIFO, a type system for information flow security. We transform the typing rules into refinement rules to get a constructive CbC approach. The approach is implemented in CoRC and evaluated by constructing several case studies to show feasibility.

In [Chapter 4](#), we conclude the thesis and give an outlook to future work. We discuss directions in which further research can be conducted.



# 2. Background

In this chapter, we introduce the relevant background necessary to understand the main contributions of this thesis. The core of this thesis is to construct functionally correct and secure programs. Therefore, the aim of this chapter is to briefly explain the thematic building blocks of this context. In [Section 2.1](#), we explain how programs on the level of methods can be specified to verify their functional correctness afterwards. In [Section 2.2](#), we introduce correctness-by-construction (CbC). CbC is an incremental program construction process that guarantees the correctness of the program under development in each refinement step. Finally, in [Section 2.3](#), we present the concept to specify and to analyze information flow properties in programs.

## 2.1. Contracts and Contract-based Verification

To prevent defects in code, the functional behavior of methods is specified with *contracts*, and a program verifier is used to prove that the method satisfies this contract. Contracts are introduced in [Section 2.1.1](#). In [Section 2.1.2](#), program verification to prove the correctness of specified methods is explained.

### 2.1.1. Method Contracts

To analyze the correctness of software, Hoare [1969] introduced *Hoare logic*, which is a formal reasoning system to specify and verify functional correctness of programs. The functional behavior of a program is defined with a *Hoare triple*  $\{P\}S\{Q\}$  with the following interpretation: when a program is in a state that satisfies the logical condition  $P$  (i.e., the precondition), then the execution of program  $S$  terminates, and the program is in a state that satisfies logical condition  $Q$  (i.e., the postcondition). The pre-/postcondition pair  $P$  and  $Q$  of a Hoare triple are logical formulas in first-order logic that specify the state of the program before and after the program is executed. This formal documentation of the program behavior can be analyzed. Specifications are used as input for test case generation or translated to assertions for runtime checks. Our focus is formal verification whether a program satisfies its specification. Formal verification addresses one shortcoming of software testing. While a test can only define the expected outcome for a concrete input, verification considers all inputs and all possible program states by formal reasoning.

The pre-/postcondition pair is also called contract [Hoare 1969; Liskov and Guttag 1986; Meyer 1992] of program  $S$ . *Design-by-contract* [Meyer 1992] is a software development paradigm that uses contracts to correctly implement object-oriented software. Design-by-contract was first introduced in connection with the programming language Eiffel. A method is specified with a pre-/postcondition contract and loop invariants. Additionally, class invariants guarantee that the specified state of a class is met before and after each method execution. Contracts are used to communicate between a caller of a method and the called method, the callee. The caller must ensure that the precondition of the callee is fulfilled. The callee then provides a result that fulfills the postcondition.



There exist numerous specification languages both inherent to a programming language and as an external add-on. For example, there exist Spec# [Barnett et al. 2004; Barnett et al. 2011] for C# programs, and the Java Modeling Language (JML) [Leavens et al. 2006] for Java programs. In this thesis, we focus on Java programs, therefore, we introduce JML with an example in the following. We focus on the JML constructs that are relevant for this thesis.

---

```

1  class Search {
2      /*@ public normal_behavior
3          @ requires a!=null && a.length>0;
4          @ ensures \result>=0 ==> a[\result]==x;
5          @*/
6      public static int linearSearch(int [] a, int x) {
7          int i = a.length-1;
8          /*@ loop_invariant i>=-1 && i<a.length
9              @ && !(\exists int k; k>i && k<a.length; a[k]==x);
10             @ decreases i+1;
11             @*/
12         while (i>=0 && a[i] != x) {
13             i = i - 1;
14         }
15         return i;
16     }
17 }

```

---

Listing 2.1.: An implementation of a linear search algorithm specified with JML

In JML, classes and methods are specified in a Java-like syntax. In Listing 2.1, a specified method `linearSearch` is shown. The method has as arguments an array of integers `a` and an integer `x`. The method computes and returns the index of the last element that has the same value as parameter `x`, or `-1` if there is no such element in array `a`. The specification of the method is a comment directly above the method. Each JML line starts with the `@` symbol to distinguish standard comments from JML. The first JML keyword `public` indicates the visibility of the specification similar to the visibility of methods in Java. The behavior of the method can be `normal_behavior` or `exceptional_behavior`. With `normal_behavior`, the method returns normally (i.e., without throwing an exception) if the precondition is fulfilled. With `exceptional_behavior`, the thrown exception for each precondition must be specified. In this thesis, we do not examine exceptional behavior, so we focus on normal executions. The keyword `requires` specifies the precondition of the method. In the example, the array `a` is required to contain at least one element. The keyword `ensures` introduces the postcondition. If the result of the method is bigger than zero, then this is the index of element with the same value as variable `x`. The case that the element is not found, is not specified explicitly. With `\result`, JML refers to the return value of the method. From lines 8 to 10, a loop invariant specifies the behavior of a Java `for-` or `while-`loop. The loop invariant must be valid before and after each loop iteration. The first part of the loop invariant specifies that the variable `i` is between `-1` and the length of the array. We need as lower bound `-1`, because if the variable `x` is not found, variable `i` is set to `-1`. The second part of the loop invariant specifies the already searched part of the array. All elements with an index between `i` and the length of the ar-



ray are not equal to  $x$ . To express this invariant, existential quantification is used. JML also supports universal quantification. With the keyword `decreases` a variant is specified. The variant is a variable that decreases monotonically with each iteration, but it is bounded from below. Thus, the termination of the loop can be guaranteed.

### 2.1.2. Program Verification

In the previous section, we introduced method contracts to specify the functional behavior of method. In this section, we explain how satisfaction of contract and implementation can be formally verified. We introduce the prominent technique of *program verification* [Schumann 2001; Ahrendt et al. 2016]. Besides this, *model-checking* and *program analysis* (e.g., dataflow analyses and abstract interpretation) are other verification techniques [Clarke et al. 2018].

*Program verification* is a deductive software verification technique. A proof certifies the correctness of a given program with respect to its specification. A formal specification and a program are translated to logical formulas, and a theorem prover applies inference rules to prove the validity of the formulas [Chang and Lee 2014; Schumann 2001]. Theorem proving can be interactive with tools, such as COQ [Coq n.d.], AGDA [Agda n.d.], LEAN [Moura et al. 2015], and ISABELLE/HOL [Nipkow et al. 2002]. Here, inference rules in a defined tactic language are applied to the proof problem until the validity is shown. There also exist automatic theorem provers, such as VAMPIRE [Kovács and Voronkov 2013] and SPASS [Weidenbach et al. 2009]. These tools synthesize complete proofs automatically by executing proof-search techniques. Of course, automatic theorem proving is limited by heuristics and does not guarantee to find a proof.

In this thesis, we focus on the automatic verification of formally specified programs. Programming languages with integrated specification language and program verifier are for example SPARK [Amtoft et al. 2008] and Dafny [Leino 2010]. An automatic program verifier for Java/JML is OpenJML [Cok 2011]. KEY [Ahrendt et al. 2016] is an interactive program verifier for Java/JML, but it offers a good automation.

**Program Verification with KEY.** The program verification tool KEY verifies Java programs specified with JML [Ahrendt et al. 2016]. The logic used by KEY to reason about programs is Java dynamic logic (JavaDL). JavaDL is an instance of dynamic logic [Harel et al. 2000] for Java. Dynamic logic [Harel et al. 2000] is an extension of first-order logic with programs. To prove the correctness of a program  $S$  specified with a precondition  $P$  and a postcondition  $Q$ , the proof problem is written in dynamic logic  $P \rightarrow \langle S \rangle Q$ . The program modality  $\langle S \rangle$  refer to the final state of program  $S$  and can be placed in front of any formula. The formula  $\langle S \rangle Q$  defines that the execution of program  $S$  terminates in a state in which  $Q$  holds. For partial correctness, the program modality  $[S]$  is defined that does not demand termination.

The first step of KEY to verify a method specified with JML is the translation from Java/JML to JavaDL. All preconditions starting with `requires` are composed to one precondition in JavaDL. The same applies for the postcondition. Implicit assumptions in JML must be stated explicitly (e.g., whether parameters or fields cannot be null). Additionally, the Java heap is modeled with read and write operations. In total, the JavaDL formula is:  $P \rightarrow \{heapAtPre := heap\} \langle self.m(args) \rangle (Q \ \& \ frame)$ . The precondition  $P$  and postcondition  $Q$  are the JavaDL condi-

tions translated from JML. The variable *heap* models the heap. By assigning  $heapAtPre := heap$ , the state of the heap before execution is saved. The *frame* is a condition that states which variables can be altered in the program. In this simplified formula, we exclude exceptions in the program execution.

KEY uses a *proof calculus*, the *sequent calculus* [Gentzen 1935], to prove the validity of the dynamic logic formula. It is based on sequents of the form  $\psi_1, \dots, \psi_n \vdash \phi_1, \dots, \phi_n$  with dynamic logic formulas  $\psi_i$  and  $\phi_i$ . With proof rules, the formula can be altered to show its validity. A proof rule defines a *conclusion* that can be drawn from *premises*. A concatenation of proof rules results in a proof tree, where the root (bottom) of the inferences is the formula to prove. This formula is valid exactly if all leaves (premises) of the tree are valid. To reason about the program in a dynamic logic formula, symbolic execution is used. Symbolic execution executes the program in natural forward direction by rewriting the program stepwise into updates of the program state [Ahrendt et al. 2016].

## 2.2. Correctness-by-Construction

*Correctness-by-Construction* (CbC) [Morgan 1994; Dijkstra 1972; Kourie and Watson 2012; Wirth 1971; Gries 1987] is a technique to incrementally construct functionally correct programs from a pre-/postcondition specification. Starting with an abstract statement, the program is constructed by refining the abstract statement stepwise into concrete program statements with so called refinement rules. Each refinement rule is proven to preserve the correctness of the program under side condition for their applicability. The process stops when a completely refined program is obtained (i.e., abstract statements no longer exist). The specification is given as a Hoare triple  $\{P\}S\{Q\}$ . The statement *S* is abstract in the beginning and concretized by applying the refinement rules to a complete program fulfilling the pre-/postcondition specification *P* and *Q*. A refinement replaces a Hoare triple  $\{P\}S\{Q\}$  with a Hoare triple  $\{P'\}S'\{Q'\}$ , where the specification and the program can be altered, but the refined Hoare triple still satisfies the starting specification. Thus, each refinement step is correct-by-construction, and leads to a correct implementation in the end.

In [Kourie and Watson 2012], the concrete program statements are written in the guarded command language [Dijkstra 1975]. The guarded command language is a Turing complete programming language with the following statements:

**Skip** A skip statement does not alter the program state.

**Abort** An abort statement is an undefined instruction that can do anything (e.g., it can loop forever).

**Assignment** The assignment statement ( $x := E$ ) assigns some expression *E* to a variable *x*. The variable and the expression must have the same type.

**Composition** With composition statement ( $S_1; S_2$ ) two statements are composed.

**Selection** The selection statement (**if**  $G_1 \rightarrow S_1$  **elseif**  $\dots G_n \rightarrow S_n$  **fi**) is a list of guarded statements of which one is chosen to execute. At least one of the guards must be evaluated to true, otherwise the selection executes **abort**. If more than one guard is true, a statement whose guard is true is nondeterministically selected.

**Repetition** The repetition statement (**do**  $G \rightarrow S$  **od**)<sup>1</sup> executes a statement as long as the loop guard is evaluated to true. Therefore, the statements can be executed zero or arbitrary many times.

## Refinement Rules

Formally, we want to find a concrete implementation  $C$  that satisfies the Hoare triple specification, written as  $Sat(C, \{P\}S\{Q\})$  [Kourie and Watson 2012]. The predicate  $Sat$  states that program  $C$  satisfies the Hoare triple  $\{P\}S\{Q\}$ . Refining a Hoare triple  $\{P\}S\{Q\}$  to a Hoare triple  $\{P'\}S'\{Q'\}$  is written as  $\{P\}S\{Q\} \sqsubseteq \{P'\}S'\{Q'\}$  with the symbol  $\sqsubseteq$  for the refinement. The refinement is correct if and only if  $\forall C : Sat(C, \{P'\}S'\{Q'\}) \rightarrow Sat(C, \{P\}S\{Q\})$ . A refinement  $\{P'\}S'\{Q'\}$  is correct if and only if every concrete program  $C$  that satisfies the refined Hoare triple must also satisfy the Hoare triple  $\{P\}S\{Q\}$ . In Definition 2.1, we present the eight refinement rules of CbC by Kourie and Watson [2012].

### Definition 2.1: Refinement Rules for the Correctness-by-Construction Approach

Let  $P$  be the precondition,  $Q$  be the postcondition, and  $S$  be an abstract statement. Then, the Hoare triple  $\{P\}S\{Q\}$  is **refinable** to

- **Skip:**  $\{P\}\mathbf{skip}\{Q\}$  iff  $P$  implies  $Q$
- **Assignment:**  $\{P\}\mathbf{x} := E\{Q\}$  iff  $P$  implies  $Q[\mathbf{x} := E]$
- **Composition:**  $\{P\}S_1; S_2\{Q\}$  iff intermediate condition  $M$  exists such that  $\{P\}S_1\{M\}$  and  $\{M\}S_2\{Q\}$  hold
- **Selection:**  $\{P\}\mathbf{if} G_1 \rightarrow S_1 \mathbf{elseif} \dots G_n \rightarrow S_n \mathbf{fi}\{Q\}$  iff  $P$  implies  $G_1 \vee \dots \vee G_n$  and  $\forall i \in \{1 \dots n\} : \{P \wedge G_i\}S_i\{Q\}$  holds
- **Repetition:**  $\{P\}\mathbf{do} [I, V] G \rightarrow S \mathbf{od}\{Q\}$  iff  $P$  implies  $I$  and  $I \wedge \neg G$  implies  $Q$  and  $\{I \wedge G\}S\{I\}$  holds and  $\{I \wedge G \wedge V = V_0\}S\{I \wedge 0 \leq V < V_0\}$  holds
- **Weaken precondition:**  $\{P'\}S\{Q\}$  iff  $P$  implies  $P'$
- **Strengthen postcondition:**  $\{P\}S\{Q'\}$  iff  $Q'$  implies  $Q$
- **Method Call:**  $\{P\}\mathbf{m}(a_1, \dots, a_n) \rightarrow b\{Q\}$  iff method  $\{P'\}\mathbf{m}(p_1, \dots, p_n) \rightarrow r\{Q'\}$  exists and  $P$  implies  $P'[p_i \setminus a_i]$  and  $Q'[\mathbf{old}(p_i) \setminus \mathbf{old}(a_i), r \setminus b]$  implies  $Q$

[Runge et al. 2019a; Kourie and Watson 2012]

**Skip Rule.** The first rule *Skip* replaces an abstract statement  $S$  with a skip statement that does not alter the program state. This refinement is correct if the precondition  $P$  implies the postcondition  $Q$ . That means, the program is already in a state that fulfills the postcondition.

<sup>1</sup>In Dijkstra's work, multiple clauses are allowed, as in the selection statement. We reduced it to only one clause for the sake of comprehensibility, without losing expressiveness.

**Assignment Rule.** The *Assignment* refinement rule introduces an assignment of an expression  $E$  to a variable  $x$ . This refinement is correct if the precondition implies the postcondition in which the variable  $x$  is replaced by the expression  $E$  [Kourie and Watson 2012].

**Composition Rule.** The *Composition* refinement rule is used to split the program in two parts with an intermediate condition  $M$ . The intermediate condition  $M$  should be fulfilled after executing a statement  $S_1$ , then statement  $S_2$  is executed to obtain postcondition  $Q$ . Both Hoare triples  $\{P\}S_1\{M\}$  and  $\{M\}S_2\{Q\}$  can be further refined by applying refinement rules.

**Selection Rule.** The *Selection* refinement rule introduces a conditional statement, where a statement  $S_i$  is executed if the guard  $G_i$  is evaluated to true. For the rule to be applicable, the precondition  $P$  must imply the disjunction of all guards  $G_i$ . This is necessary such that at least one statement  $S_i$  can be executed and the program does not get stuck. The refined Hoare triples  $\{P \wedge G_i\}S_i\{Q\}$  contain the guards as part of their preconditions.

**Repetition Rule.** The *Repetition* refinement rule introduces a loop statement. The statement  $S$  is executed as long as the guard  $G$  is evaluates to true. To guarantee the correctness of the loop, we specify a loop invariant  $I$ , a condition that is true at the start and end of each iteration of the loop, and a loop variant  $V$ , an integer expression that decrease monotonically in each iteration. The side condition of the refinement rules has four parts: (1) the precondition  $P$  must imply the invariant  $I$  so that the invariant is fulfilled before the first loop iteration; (2) the conjoined invariant and the negated guard  $G$  must imply the postcondition  $Q$ . With this implication, it is ensured that the postcondition is fulfilled after the last loop iteration; (3) the loop body must preserve the invariant; (4) the variant must decrease monotonically, but it is bounded from below by  $0$ ; The expression  $V_0$  points to the value of the variant before a loop iteration to compare it with the value afterwards. With these conditions, the correctness and termination of the loop is ensured.

**Weaken Precondition Rule.** The precondition of a Hoare triple is weakened with the *Weaken precondition* rule. This rule is applicable if  $P$  implies  $P'$ . Every program  $C$  that satisfies the postcondition from the weaker precondition  $P'$  also fulfills the starting Hoare triple.

**Strengthen Postcondition Rule.** The postcondition of a Hoare triple is strengthened with the *Strengthen postcondition* rule. This rule is applicable if  $Q'$  implies  $Q$ . Every program  $C$  that satisfies the stronger postcondition  $Q'$  also satisfies the weaker postcondition  $Q$ .

**Method Call Rule.** The *Method Call* refinement rule introduces a method call. A method  $m(p_1, \dots, p_n) \rightarrow r$  must exist with method name  $m$ , formal parameters  $p_1, \dots, p_n$ , return value  $r$ , precondition  $P'$ , and postcondition  $Q'$ . We assume that the parameters are passed with *call-by-value* to exclude side effects [Bordis et al. 2020b]. When the method is called, actual values  $a_1, \dots, a_n$  are passed to the method and a return value  $b$  is computed. For the method call to be correct, the precondition  $P$  must imply the precondition  $P'$  of the method but the formal parameters  $p_i$  are replaced with the actual parameters  $a_i$ . This implication ensures that the precondition of the method is satisfied. Similar, we ensure that the method's postcondition satisfies the postcondition

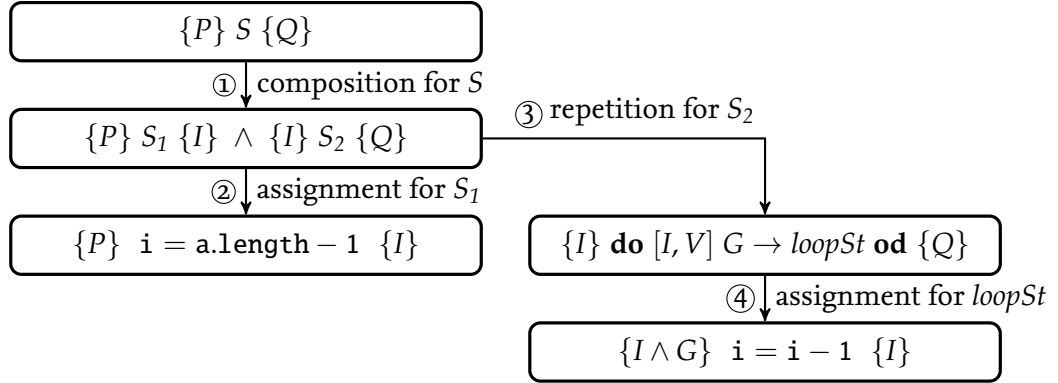


Figure 2.1.: Refinement steps for the linear search algorithm

of the refined Hoare triple. The postcondition  $Q'$  must imply the postcondition  $Q$ . In  $Q'$ , we refer to the parameters before executing the method with  $\text{old}(p_i)$  because the values of the parameters can change in the body of the method. As for the precondition, the formal parameters are replaced with the actual parameters. The same applies for the formal and actual return value. To be correct, the return value must not be used in the precondition because it is not yet computed. The parameters  $p_i$  must not be used in the postcondition because their scope is limited to the method's body.

**Example 2.1.** In [Figure 2.1](#), we give an example of a program developed with CbC. The linear search algorithm of [Listing 2.1](#) is constructed. We use the same pre-/postcondition specification for the algorithm as before. The precondition states that the array has at least one element ( $P := a \neq \text{null} \ \& \ a.\text{length} \geq 0$ ). In the postcondition, we return the index  $i$  where the (last) element  $x$  was found ( $Q := i \geq 0 \ \& \ a[i] = x$ ), or a negative integer if the element is not found.

The linear search algorithm traverses the array from the end to the start and checks for each index whether the current element is equal to the searched variable  $x$ . If an element with the same value is found, the index is returned. If variable  $x$  is not found,  $-1$  will be returned. We use the same invariant as above. To shorten the invariant in the CbC program, the predicate `appears` is used: `appears(a, x, l, h)` asserts that in the array  $a$ , the variable  $x$  occurs between the indices  $l$  (included) and  $h$  (excluded). In the invariant ( $I := \text{!appears}(a, x, i + 1, a.\text{length}) \ \& \ i \geq -1 \ \& \ i < a.\text{length}$ ) the predicate is negated to determine the already searched part of the array where the variable  $x$  does not occur.

To construct the algorithm, we apply refinement rules. For the rules to be applicable, side conditions are checked (see for each rule the conditions after the *iff* in [Definition 2.1](#)). Hoare triples that contain an abstract statement (e.g., the two Hoare triples after applying the composition refinement rule) can be further refined and are checked as soon as the abstract statement is concretized. A Hoare triple with a concrete statement is a leaf of the refinement chain. These are assignment, skip, or method call statements.

The first refinement ① uses the composition rule to split the program into two statements with an intermediate condition that is the invariant  $I$ . The first statement will be refined to initialize the loop variable  $i$  and the second statement will be refined to the loop. We start at the end of array, by refining the first statement to  $i = a.\text{length} - 1$  with the assignment rule ②. The assignment refinement fulfills its side condition because the intermediate condition is satisfied (`!appears(a, x, a.length, a.length)`). The `appears` predicate dissolves to an existential quantification with an empty set that always evaluates to false, but this predicate is negated in the postcondition.



The second statement is refined with the repetition rule ③. The invariant is the same as the intermediate condition, therefore we know that the invariant is fulfilled before the first loop iteration. The loop is executed as long as the variable  $x$  is not found and the start of the array is not reached ( $G := i \geq 0 \ \& \ a[i]! = x$ ). That the repetition rule is correctly applied, not only the invariant must be fulfilled in the beginning, but also the negated guard and the invariant must imply the postcondition  $(!(i \geq 0 \ \& \ a[i]! = x) \ \& \ I) \rightarrow (i \geq 0 \rightarrow a[i] = x)$ . With known transformations, it can be shown that the negated guard and the postcondition are equal, therefore the implication is true.

In the last step ④, the loop body is refined with the assignment rule to  $i = i - 1$ . We check that the invariant is preserved in each iteration. Assumed that the loop guard and the invariant hold, the invariant is fulfilled after executing the assignment. We also guarantee that the loop terminates because the variant ( $V := i + 1$ ) is decreased in each iteration and bounded from below (i.e., the variant is never smaller than zero). As no abstract statement is left in the CbC program, the incremental program construction stops and the resulting implementation of the linear search algorithm is guaranteed to be correct-by-construction. In comparison to post-hoc verification, we have additional specification overhead by giving the intermediate condition (both approaches still need the loop invariant), but the stepwise construction supports finding defects, since correctness guarantees are given for each applied refinement.

## 2.3. Information Flow Control

Information flow control [Sabelfeld and Myers 2003] is used to establish confidentiality and integrity of program data. The standard way to ensure confidentiality and integrity is to analyze the flow of information from sources to sinks in the software. Confidentiality means that an information flow of private data to insecure destinations must be prevented. Similarly, integrity ensures that secure systems are not influenced by untrusted sources [Biba 1977; Sabelfeld and Myers 2003]. This allowed and prohibited flow of information is specified in an information flow policy. An information flow policy specifies the allowed and prohibited flow of information between variables labeled with a security level. For example, an information flow from public to secret variables is allowed, but the other direction is prohibited. Security levels are often referred to as low for public and high for secret. Information can flow *directly* by assigning a value to a variable, or information can flow *indirectly*. For example, by observing a taken conditional branch, an attacker may be able to draw conclusions about the value of the guard.

Bell and La Padula [1976], and Denning [1976] introduced lattices of security levels to state information flow policies. A Lattice is defined in Definition 2.2, it arranges the security levels and shows the allowed flow through directed edges.

**Example 2.2.** We give some examples of secure and insecure program statements in Listing 2.2. We assume the information flow policy from above with low and high security levels and an allowed flow from low to high. The assignment in Line 2 is secure. A low expression can be assigned to a high variable. In contrast, the assignment in Line 3 is not secure because a variable of security level low can only store expressions of the same security level or lower. The next three examples in lines 5, 8, and 11 show indirect information flows. To prevent leaks through indirect information flows, assignments in the branches of the conditional-statements must have at least the security level of the expression in the guard. Therefore, the conditional statement in Line 5 is secure. A low guard is used and this does not restrict the assignments in the branches. Both assignments to

**Definition 2.2: Bounded Upper Semi-Lattice (for Information Flow Policies)**

A bounded upper semi-lattice is a structure  $\langle L, \leq, lub, \top, \perp \rangle$  where the security levels  $L$  are partially ordered with  $\leq$ ,  $lub$  is the least upper bound operator, and  $\top$  and  $\perp$  are the top and bottom security level of the lattice. The  $\leq$  operator is reflexive, antisymmetric, and transitive, but as it is a partial order, not all elements must be comparable. An upper bound operator defines for a set of elements  $X \subseteq L$  that an upper bound  $y \in L$  exists if  $\forall x \in X : x \leq y$ . The least upper bound ( $lub$ ) is a security level  $u \in L$  for that holds: for a set of elements  $X \subseteq L$  if  $u \leq y$  for all upper bounds  $y$  of the set  $X$ . To be an upper semi-lattice, a unique least upper bound for every subset  $X \subseteq L$  must exist. The lattice is bounded by the top element  $\top$  and the bottom element  $\perp$ . This means  $\forall l \in L : \perp \leq l \leq \top$ .

```

1 public void exampleMethod(low int l, high int h) {
2     h = 1; //ok, flow from low to high
3     l = h; //wrong, direct flow from high to low
4
5     if (l>0) {l = 1;} else {h = 2;}
6     //ok, high can be used in the branches
7
8     if (h>0) {l = 1;} else {h = 2;}
9     //wrong, indirect flow from high to low
10
11    if (h>0) {h = 1;} else {h = 2;}
12    //ok, only high is used
13 }

```

Listing 2.2.: Examples of of well-typed and ill-typed statements

*low or high variables are secure. In Line 8, a high guard restricts the branches to assign to high variables. This line is insecure because an assignment to a low variable  $l$  exists. By reading the value of  $l$ , it can be deduced whether variable  $h$  is greater than zero. Line 11 is secure because there are only assignments to high variables.*

**Type System for Secure Information Flow.** In this thesis, we focus on type systems as a language-based technique [Sabelfeld and Myers 2003] to reason about information flow security. A type system labels every variable and expression with a security type, and typing rules check that assignments of security types do not violate the security policy. In Figure 2.2, we show an excerpt of the security type system of Volpano et al. [1996] that can type-check a simple imperative language. The language contains statement  $c$  and expressions  $e$ . Statements are typed with  $\tau \text{ cmd}$ , expressions are typed with  $\tau$ , and variables are expressions that are typed with  $\tau \text{ var}$ , where  $\tau$  is a security level of the lattice. Both expressions and statements are phrases  $p$ . The security types  $\tau$ ,  $\tau \text{ cmd}$ , and  $\tau \text{ var}$  are phrase types  $\rho$ . The security type on expressions is used to analyze direct information flow, and the security type on statements is used to analyze indirect information flow. The first typing rule types a variable  $x$  with  $\tau \text{ var}$ . The security types are arranged in a lattice structure with the operator  $\leq$  to compare

$$\begin{array}{c}
\vdash x : \tau \text{ var} \quad (1) \qquad \frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'} \quad (2) \qquad \frac{\vdash p : \rho \quad \vdash \rho \subseteq \rho'}{\vdash p : \rho'} \quad (3) \qquad \frac{\vdash \tau \subseteq \tau'}{\vdash \tau' \text{ cmd} \subseteq \tau \text{ cmd}} \quad (4) \\
\frac{\vdash x : \tau \text{ var} \quad \vdash e' : \tau}{\vdash x := e' : \tau \text{ cmd}} \quad (5) \qquad \frac{\vdash c : \tau \text{ cmd} \quad \vdash c' : \tau \text{ cmd}}{\vdash c; c' : \tau \text{ cmd}} \quad (6) \qquad \frac{\vdash e : \tau \quad \vdash c : \tau \text{ cmd}}{\vdash \mathbf{while} \ e \ \mathbf{do} \ c : \tau \text{ cmd}} \quad (7) \\
\frac{\vdash e : \tau \quad \vdash c : \tau \text{ cmd} \quad \vdash c' : \tau \text{ cmd}}{\vdash \mathbf{if} \ e \ \mathbf{then} \ c \ \mathbf{else} \ c' : \tau \text{ cmd}} \quad (8)
\end{array}$$

Figure 2.2.: Security type system [Volpano et al. 1996]

security types. This order is extended to a subtype relation with rules 2–4. The Rule 5 determines a secure assignment of an expression  $e'$  to a variable  $x$ . The security types must comply, but subtyping can be used to assign, for example, a low expression to a high variable. Rule 6 concatenates two statements with the same security type. Rules 7 and 8 type loop and conditional statements. Here, the security type of the guard  $e$  and the statements  $c$  and  $c'$  must comply.

**Example 2.3.** *The statements that comply with the security policy of the example method in Listing 2.2 are typable with this type system. In Line 2, subtyping is used to coerce the type of the expression to high, so that it can be assigned to the variable with the high type. An assignment of a higher expression to a lower variable is not typable in Line 3. A statement  $c$  has a type  $\tau \text{ cmd}$ . This type ensures that an assignment in statement  $c$  is made to a variable with the security type  $\tau \text{ var}$ . The conditional statement in Line 5 has a low guard. Therefore, the assignments in the branches have to be of type low. Both assignments in the branches are typable because the branches can be coerced to a low security type since the subtype relation for statements is contravariant. In lines 8 and 11, a high guard restricts the statement to have a high type. Therefore, Line 11 is typable as only high assignments are used, but Line 8 is not typable. In Line 8, the low statement in the then branch of the conditional-statement violates Rule 8 to have the same high type as the guard.*



# 3. Contributions

In this thesis, we enable and support a correct-by-construction software development process for functionally correct and secure programs. The contributions of the thesis are grouped into four parts and answer the four research questions **RQ1** – **RQ4**. The first contribution implementing tool support for CbC addresses **RQ1**. The second contribution compares CbC with post-hoc verification in two user studies to address **RQ2**. The third contribution compares CbC by Kourie and Watson [2012] with related CbC approaches and addresses **RQ3**. The related CbC approaches are introduced and discussed in terms of their language constructs for more flexible program construction in comparison to CbC. The fourth contribution introduces CbC for information flow security and addresses **RQ4**. Refinement rules are developed that guarantee compliance with defined information flow policies. The approach is implemented in CoRC to support a uniform CbC approach for functional correctness and security.

## 3.1. CoRC Tool Support for Functional Correct-by-Construction Program Development

With our first contribution, we address the research question RQ1: *How can we support a correct-by-construction program development approach for functionally correct software?* Correctness-by-construction by Kourie and Watson [2012] is considered a viable approach for developers, but tool support is missing. Therefore, we present how our open-source development environment, named CoRC,<sup>1</sup> supports correctness-by-construction by Kourie and Watson [2012] and guarantees the correctness of programs under development. Our work was presented at the International Conference on Fundamental Approaches to Software Engineering in 2019 (see Section A.1).

T. Runge, I. Schaefer, L. Cleophas, T. Thüm, D. Kourie, and B. W. Watson [2019a]. “Tool Support for Correctness-by-Construction”. In: *International Conference on Fundamental Approaches to Software Engineering*. Vol. 11424. Lecture Notes in Computer Science. Springer, pp. 25–42. DOI: [10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2)

### Requirements for CbC Tool Support

According to our vision, our goal is to establish CbC as program development approach and provide an alternative to post-hoc verification. This means that adequate tool support should naturally exhibit a high quality in usability, and the approach itself must be presented to developers in a way that maximizes comprehension. We also need a scalable approach so that case studies of reasonable size can be created. Correctness-by-construction by Kourie and Watson [2012] promises

<sup>1</sup><https://github.com/KIT-TVA/CoRC>

these properties, but no tool support exists yet. Therefore, we derived requirements from the desired CbC development process in consultation with Kourie and Watson to implement an appropriate tool. The listed requirements are specific for correctness-by-construction. For clarity, we structure the requirements into three categories. First, we present requirements to refine programs ( $R_0$ ,  $R_1$ ), second, requirements for correctness checks ( $P_0$ – $P_3$ ), and third, requirements for the IDE, GUI, editing capabilities, and scalability ( $G_0$ – $G_3$ ).

To create programs following the CbC approach, the tool has to support all CbC refinement rules of Kourie and Watson [2012] ( $R_0$ ). The user is guided by these rules to programs which are correct by construction. We argue that the programs should be written in an established language such that developers are willing to use the tool. An established language also supports comprehensibility for the developers. Additionally, a concept of abstract statements is necessary which are refinable to concrete implementations. Abstract statements are the main concept to indicate parts of the program that can be further refined. To specify the program, pre- and postconditions and invariants have to be written in a specification language ( $R_1$ ) which should be checkable by machines. The specification language must support first-order logic as needed for the CbC methodology by Kourie and Watson [2012].

The CbC methodology guarantees the correctness of developed programs. Therefore, the tool should verify each refinement step to support this property. The side conditions of each refinement rule must be proved by a program verifier in order to show partial correctness ( $P_0$ ). Total correctness is proved by showing that every repetition statement terminates ( $P_1$ ). Here, it must be proved that the loop variant decreases in each loop iteration. The proof results have to be stored and linked with the editor to track the status of the verification for the developer ( $P_2$ ). If something changes in the program, the affected refinement steps have to be reproved. Verifications still to be performed should be displayed to the developer so that the developer can investigate these proof obligations. Detailed feedback is required for good usability of the tool.

The IDE can offer a graphical ( $G_0$ ) and a textual ( $G_1$ ) editor to support different kinds of development styles. In the graphical view, a tree structure can visualize the refinement steps where Hoare triples are nodes and refinement steps are edges. The program is refined top-down from a starting Hoare triple with an abstract statement to Hoare triples in the leafs with concrete statements. The graphical view is especially interesting for teaching this alternative programming style as refinement steps and the verification status are explicitly shown. The textual editor should be a plain textual editor to write CbC programs enriched with annotations for specifications. This editor is intended for users who want a “familiar coding feeling”. The textual editor is useful to develop programs with known features of textual IDEs, such as syntax highlighting, auto-completion, or copy and paste. To not limit users to one view, we need automatic switching between both views ( $G_2$ ). The user should be able to write the program in one view and generate the other view automatically. Additionally, the IDE should support the development of meaningful programs ( $G_3$ ). The development of programs with CbC must be integrated into a holistic program development process such that the tool is usable to develop case studies. An integration of the tool into an IDE is necessary, so the developer gets an overview of all projects, can open, close, and save different programs, starts verification attempts, and gets comprehensible error messages.

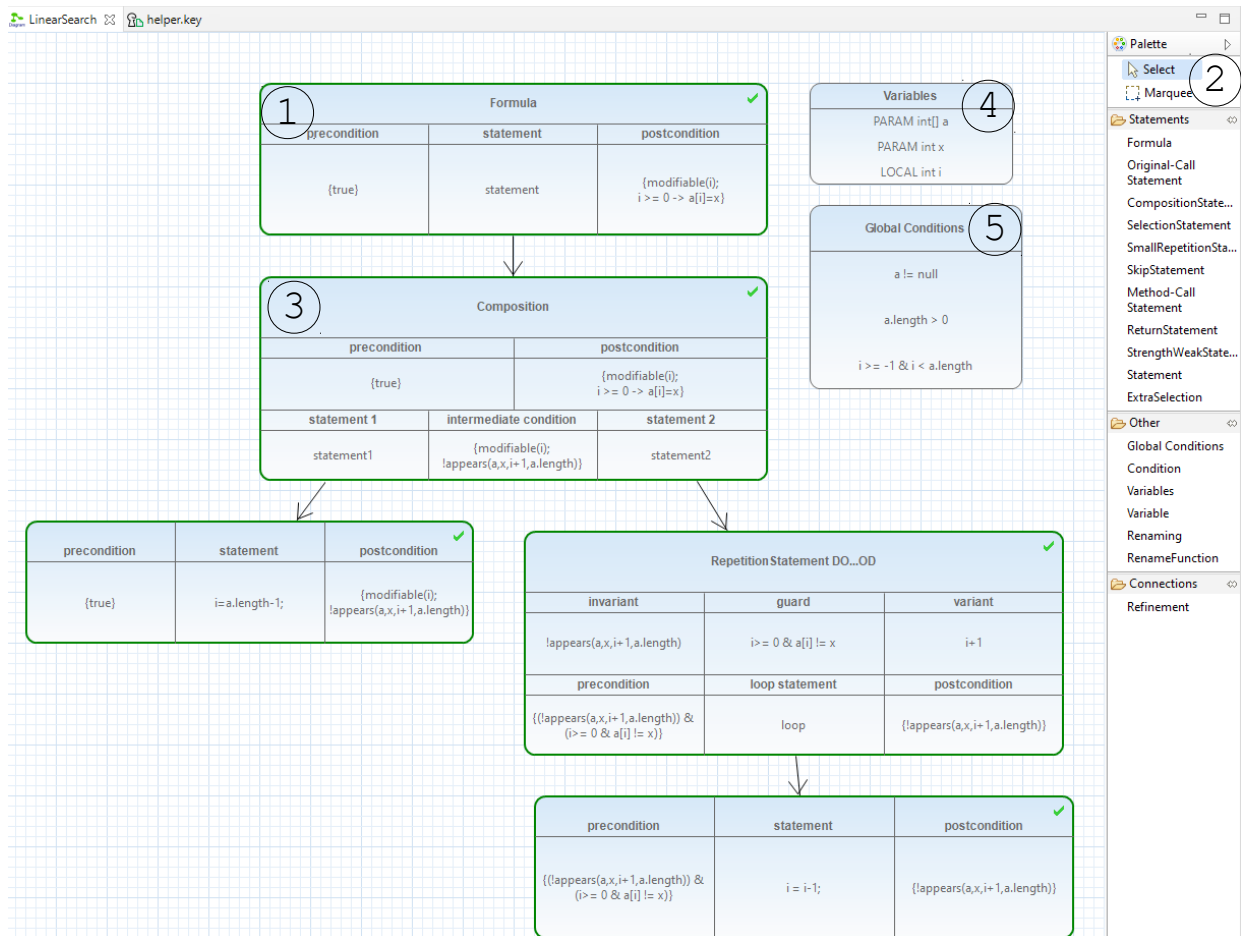


Figure 3.1.: Linear search algorithm in the graphical IDE

## CORC Tool Support

CORC is an open-source IDE that implements the refinement-based program development approach given by Kourie and Watson [2012]. Programs in CORC are written in Java. We chose Java because object-oriented programming is widely used and Java is a known representative of this category. By choosing Java we can build on the well-established specification language JML [Leavens et al. 1998] and verification tools for Java/JML, such as KEY [Ahrendt et al. 2016].

The program development in CORC starts with a Hoare triple specification. Then, refinement rules are applied to construct a concrete implementation in Java. CORC supports the refinement rules of Kourie and Watson [2012] presented in Section 2.2. Each applied refinement is guaranteed to be correct by proving side conditions of the refinement rules. The side conditions are translated to proof obligations that are discharged by the program verifier KEY. KEY automatically returns the result whether the applied refinement can be proved correct. By integrating KEY in CORC, we build on an established program verifier for Java code that is extensively tested, which increases trust in the verification results. Although we rely on KEY to discharge our proof obligations, the correctness-by-construction approach is not tailored to a specific programming or specification language or a specific verification tool in general.

**Example 3.1.** To explain program development in CORC, we represent the linear search algorithm of [Figure 2.1](#) in the graphical editor in [Figure 3.1](#). The graphical editor is intended to learn and understand CbC by visualizing the Hoare triple specifications, the refinements in a tree structure, and the verification status. Each node, visualized as a light-blue box, is a statement, and each applied refinement rule is an edge between the nodes. A statement is one of the program statements presented in [Section 2.2](#), such as assignment (i.e., the leaf nodes) or composition. We refer to the circled numbers in the figure. ① The starting statement of the refinement tree represents a Hoare triple that is specified with a pre-/postcondition. The developer writes the conditions in JavaDL by clicking on the text fields. Using JavaDL is a design decision to save a translation step for the verification with KEY. JavaDL and JML are similar with the same expressiveness. The differences are described by Ahrendt et al. [2016]. The abstract statement in the middle of that Hoare triple has a placeholder name which can be changed manually by the developer. This abstract statement is then refined stepwise. ② Refinements are applied via drag-and-drop from the palette on the right side. The palette on the right has a graphical representation for each program statement. By connecting an abstract statement with an added statement from the palette, the corresponding refinement rule to introduce that program statement is applied. For example, ③ the composition statement was added to the diagram and then connected with the abstract statement of the starting Hoare triple. The pre-/postcondition in the composition statement is inherited from the parent Hoare triple as defined in the refinement rule. We explicitly show the inherited conditions in the child node to improve readability so that the developer directly sees the relevant pre- and postcondition in each node. In the case of the composition rule, the developer must introduce an intermediate condition for the rule to be fully applied. Afterwards, two assignment statements and a repetition statement are introduced during program construction. In assignment statements, the developer writes the explicit assignments in Java syntax. For the repetition statement, guard, loop invariant, and loop variant are specified. ④ The parameters and local variables of the method are shown on the top right node. Parameters are extracted from the method signature of the method. Local variables are added by the developer. ⑤ The global conditions (e.g., object invariants) are also written by the developer. These conditions are valid throughout the program (e.g., the size of the array). They are added to each pre-/postcondition of a Hoare triple. This reduces writing effort to not repeat the same condition in each Hoare triple.

The green border around each node indicates that all refinement steps are proved correctly. We have to prove for each applied refinement rule the side condition (see for each refinement rule the conditions after the *iff* in [Definition 2.1](#)). This condition is translated into a proof obligation in JavaDL. Since we verify rather small problems (e.g., with just a single assignment), KEY is usually able to close the proof automatically if the JavaDL formula is valid. In CORC, the proof of the refinement steps is started on request of the developer. As each refinement step can be proved independently of the others (as long as the specification remains the same), the developer is free to prove the refinement steps in any order. If the CORC program is altered, CORC automatically tracks which refinement steps have to be reproved such that the program is always guaranteed to be correct. When a refinement step is not provable, the affected node is highlighted with a red border. With this visualization, the developer directly sees the problematic refinement steps. The developer can alter conditions, assignments statements, or change complete refinement steps, and restart the verification process.

## Implementation of CORC

The CORC tool is based on Eclipse to benefit from the standard IDE features, such as a project explorer, context menus, and a console. The language of CORC is implemented with an EMF<sup>2</sup> metamodel. The metamodel defines the concrete statements with their attributes (pre-/postconditions, abstract statements, verification status, ...) and the possibility to refine abstract statements. The editor of CORC is split into a graphical and a textual editor. Both editors are implemented based on the same metamodel to allow the interchange of programs written in one of the editors. The graphical editor is implemented with Graphiti.<sup>3</sup> Graphiti is a framework for graphical editors to define the shape and layout of graphical nodes that represent statements, as well as the functions for adding, updating, and deleting the nodes. The textual editor is implemented with Xtext.<sup>4</sup> Xtext is a framework that create a parser and a full-featured Eclipse text editor from a textual domain-specific language (DSL). We designed a DSL based on our metamodel. When a developer writes a CORC program in the textual editor, the program is parsed and translated to an instance of the metamodel to check its correctness. In both editors, proof obligation for each applied refinement rule are generated as proof-files readable by KEY. Afterwards, the verification is performed with KEY, and results are automatically returned to CORC to present and save the proof status.

## Validation of CORC

We implemented case studies to evaluate whether CORC fulfills the stated requirements and to measure the verification effort in comparison to post-hoc verification. Eight algorithms from the book by Kourie and Watson [2012] are implemented. In a further evaluation [Bordis et al. 2022a], 40 methods from the case studies BankAccount [Thüm et al. 2012], Email [R. J. Hall 2005], and Elevator [Plath and Ryan 2001] are implemented.

**Qualitative Evaluation.** By implementing case studies, we were able to evaluate that CORC is suitable to develop programs according to CbC. With CORC, we could develop programs and guarantee their correctness ( $R_0$ ,  $R_1$ ). All refinement rules of [Kourie and Watson 2012] are implemented in CORC. We can specify the CORC methods with pre-/postconditions, intermediate conditions, invariants, and variants. With Java, we support an established language. The specification languages JML and JavaDL support first-order logic as needed for CbC. In CORC, we implemented a graphical editor for the overview of all refinements in a tree structure (Go). An advantage of this editor is that all pre-/postconditions, invariants, and variants are shown explicitly next to the statement. The writing effort is reduced because conditions are propagated through the method automatically. Only newly introduced invariants, intermediate conditions, or assignments have to be written manually by the developer. The tree structure gives an overview of applied refinement steps to track possible defects in the method under construction. To not miss a proof, the border color of each node indicates the current proof status. We can also track which parts have to be proved again if the method is changed (P2). We also implemented a textual editor to create methods following the CbC approach (G1). An advantage of the editor is that auto-completion supports the developers in writing

<sup>2</sup><https://www.eclipse.org/modeling/emf/>

<sup>3</sup><https://eclipse.org/graphiti/>

<sup>4</sup><https://eclipse.org/Xtext/>

the method. As all proof obligations are generated automatically, no correctness proof is missed. If one proof cannot be closed, the specific part in the method is highlighted and can be reviewed manually (P<sub>2</sub>). The automatic switch between both views (G<sub>2</sub>) is also implemented. CoRC is integrated into the Eclipse IDE. Therefore, necessary functionality to manage CoRC projects, such as the project explorer and the console are available and integrated in CoRC. The verification of methods is done by generation proof obligations which are discharged by KEY. We ensure partial correctness by proving every side condition of the refinement steps (P<sub>0</sub>), this also includes total correctness because this is a side condition of the repetition rule (P<sub>1</sub>). With KEY as backend, we are able to verify similar methods in CoRC that can be verified post-hoc with KEY. As KEY is able to verify sufficiently complex case studies [Gouw et al. 2015], we assume that CoRC should be able to handle the same case studies (G<sub>3</sub>). To summarize, our tool support satisfies all requirements of Section 3.1, therefore we can answer that CoRC is an adequate tool support for CbC.

**Verification Effort.** We also implemented the case studies to evaluate CbC and CoRC in comparison to post-hoc verification in terms of verification effort (verification time and verification steps). In a first evaluation, we measured the verification effort by constructing and verifying eight algorithms from the book by Kourie and Watson [2012] in CoRC. We also implemented the algorithms in Java and verified them post-hoc with KEY. In this evaluation, we measured that the verification time is reduced significantly which indicates a reduced proof complexity. In a second evaluation with 40 methods from the three case studies BankAccount [Thüm et al. 2012], Email [R. J. Hall 2005], and Elevator [Plath and Ryan 2001], no trend for the verification effort can be identified [Bordis et al. 2022a]. We measured a similar verification effort for both approaches. Nevertheless, an important point is that we verified six more methods with CoRC in that second evaluation. We verified only 34 method with post-hoc verification. The additional specification in CoRC and the fine-grained proofs have a positive impact on the provability of methods, but the verification effort for verifiable methods is not significantly affected. For future work, more case studies are needed to further investigate the verification effort.

## Related Work

We discuss related work for specifying and verifying software. We also discuss related correctness-by-construction program development approaches. Here, we compare CoRC with other CbC tools.

**Contracts and Program Verification.** In CoRC, we use JML, JavaDL and Java to specify and construct programs. We integrated KEY [Ahrendt et al. 2016] for the verification tasks. KEY is designed for deductive program verification of Java/JML programs. A related tool is OpenJML [Cok 2011] that also verifies Java programs specified with JML.

First, the object-oriented programming language Eiffel supports design-by-contract [Meyer 1988; Meyer 1992] by offering contracts in the language. In Eiffel, classes are specified with invariants, and methods are specified with pre-/postconditions contracts. To verify methods, the program verifier AutoProof [Khazeev et al. 2016; Tschannen et al. 2015] is integrated. AutoProof translates the specified method to a logic formula, and an SMT-solver is called to prove the validity of this formula. In the backend, AutoProof uses Boogie [Barnett et al. 2005]. The language Spec# is an extension of C#



that introduces contracts and invariants [Barnett et al. 2004; Barnett et al. 2011]. Spec# programs are verified by translating the proof obligations to formulas verifiable with Boogie [Barnett et al. 2005]. For the C language, the program verifier VCC [Cohen et al. 2009] and Frama-C [Cuoq et al. 2012] verify specified C code. For this, VCC reuses the Spec# tool chain. The VeriFast [Jacobs et al. 2010] program verifier verifies C and Java programs specified with contracts. The contracts are written in separation logic instead of first-order logic as used in CORC. VerCors [Amighi et al. 2014] also verifies C and Java programs. This tool is focused on concurrent and distributed software. Besides Eiffel, Dafny [Leino 2010] is another language with integrated specifications and verification. Dafny is a functional language that supports the compilation to languages, such as C#, Java, Go, and Python. Whiley [Pearce and Groves 2013] is another programming and specification language with associated verifier. For a subset of Ada, the languages SPARK [Barnes 2003] supports specification and verification of Ada programs. The specification is written in the Ada *aspect*-syntax.

The focus of the presented languages is the specification of program behavior, and the focus of the presented program verifiers is the verification that specified programs satisfy their specification. In our contribution of developing the CORC tool support for CbC, we include the process to construct programs that are guaranteed to be correct in every development step, instead of just verifying the correctness post-hoc. For the verification of our proof obligations in CORC (i.e., the verification whether side conditions are satisfied), we decided to build on KEY, but the CbC approach is not limited to a specific language or verification tool. Therefore, the mentioned related languages and tools can be used to implement CbC.

**Refinement-based Correctness-by-Construction.** Correctness-by-construction is designed for the incremental construction of specified programs with correctness guarantees for each development step. Pioneers in this field are Dijkstra [1972], Wirth [1971], Gerhart [1975], and Hoare [1969]. For CORC, we implemented correctness-by-construction by Kourie and Watson [2012] that is based on Dijkstra [1976] and Gries [1987]. For Gries' program development method, Heisel [1992] proposed tool support that builds correctness proofs of the program under development. This tool is tailored to the guarded command language in contrast to CORC where programs are constructed in Java. Other related CbC methodologies are Morgan's refinement calculus [Morgan 1994] and invariant-based programming [Back and Wright 2012; Back 2009].

Morgan's refinement calculus and correctness-by-construction by Kourie and Watson [2012] have the same theoretical foundation, but Morgan's refinement calculus is more elaborated with numerous refinement rules, where many of these rules are only formally interesting. Kourie and Watson [2012] propose a reduced set of refinements rules, so that comprehensibility is improved without loosing expressiveness. The language ArcAngel [Oliveira et al. 2003] with the program verifier ProofPower [Zeyda et al. 2009] implements Morgan's refinement calculus. A tactic language is introduced, where a tactic applies a sequence of refinement rules to construct correct programs. The applied tactic discharges proof obligations that are verified with ProofPower. Invariant-based programming [Back and Wright 2012; Back 2009] focuses on invariants instead of pre-/postcondition contracts as starting point. The approach is implemented in the tool SO-COS [Back 2009; Back et al. 2007]. As CORC, SO-COS has a graphical user interface to construct programs. SO-COS uses a form of a UML-style state chart where new states and transitions are introduced in a refinement step. The tool verifies that the refinement step complies with the stated

invariants. The finished program is guaranteed to be correct and can be translated to executable code. The applied refinement steps in CoRC are structured in a hierarchical tree. This tree represents the structure of the code (comparable with an abstract syntax tree). Therefore, developers construct with CoRC at an abstraction level closer to source code.

Further refinement-based CbC approaches are Event-B [Abrial 2010; Abrial et al. 2010] for automata-based systems and Circus [Oliveira et al. 2009; Oliveira et al. 2008] for state-rich reactive systems. These approaches have an abstraction level on models (i.e., automata or state machines). These models are refined stepwise to concrete and executable implementations. Each refinement step results in a more concrete model that guarantees conformations with the initial model. The tool Rodin [Abrial et al. 2010] implements Event-B, and CRefine [Oliveira et al. 2008] implements Circus. For both approaches, the main difference to CoRC is the different abstraction level. In CoRC, specified source code is constructed instead of automata-based systems.

The C-by-C approach of A. Hall and Chapman [2002] uses formal modeling processes to guide the development during all stages (architectural design, detailed design, code) and to analyze the results. This should eliminate defects in early stages of the development. With CoRC, we provide support at the method implementation level and not a process for all design stages.

**Program and Specification Synthesis** Program synthesis is a technique for automatically generating programs based on user specifications. Pioneers in this field are Manna and Waldinger [1980]. A state-of-the-art of program synthesis approaches is given by Gulwani et al. [2017]. For the Fortran language, Stickel et al. [1994] deductively synthesize programs from user-given graphical specifications. Procedures given in libraries are composed to full implementations. Gulwani et al. [2010] also synthesize programs by assembling basic components from a specified library. The approach of Polikarpova et al. [2016] synthesizes recursive programs from specifications using type information. Inversely, synthesis of function summaries [Hoare 1971; Chen et al. 2015; Sery et al. 2012] generates pre-/postcondition specifications from programs. This synthesis of function summaries is used for modular verification to reduce verification time. With CbC and CoRC, developers construct programs in conformance with a pre-/postcondition specification. Thus, CbC is a program development approach where developers construct both, the desired program and its specification. Program synthesis generates only one of possibly many programs that fulfill the given specification, and synthesis of a function summary generates one of possibly many specifications for a program. In general, synthesis is limited in scalability due to the enormous search space of programs/specifications and the ambiguity of user intent.

## Conclusion

The goal of this work is to develop tool support for correctness-by-construction to answer *Research Question RQ1 – Tool Support for Functional Correctness-by-Construction*. We presented CoRC that enables CbC as proposed by Kourie and Watson [2012]. CoRC supports all refinement rules of Kourie and Watson [2012] and guarantees the correctness of the program in each refinement step. We support the development of programs in a graphical and a textual editor to cover different needs of developers. In general, we fulfill the stated requirements to support correctness-by-construction. Therefore, we addressed **Challenge 1** with this contribution. We also evaluated CoRC by measur-



ing the verification effort in comparison to post-hoc verification. For small algorithms, we measured reduced verification time, but we could not confirm this in further cases studies [Bordis et al. 2022a]. In these case studies, we measured a similar verification time for both approaches. With existing tool support, we can evaluate with user studies how software developers use CbC. The user studies are presented in the next section. CORC is also the foundation for the further work in this thesis; the comparison with other correctness-by-construction approaches in Section 3.3, and the integration of information flow security into the CbC approach in Section 3.4.

## 3.2. A Usability Evaluation of the CbC Approach as Implemented in CORC

To evaluate comprehensibility and usability of CbC through CORC, user studies are necessary. With the results of these user studies, we want to answer the second research question RQ2: *How usable is the correct-by-construction program development approach with CORC?* Hence, we want to evaluate whether CbC can have a positive impact on the development of correct programs (e.g., developers could state that defects are more easily detected with CbC). In this section, we present two user studies in which we evaluate how participants interact with CORC to construct correct programs in comparison to how participants implement programs in Java and verify their correctness post-hoc with KEY. By analyzing how the participants construct the programs and interact with the tools, we get insights in their programming procedure. We collect qualitative results by conducting a structured interview and a standardized user experience questionnaire.<sup>5</sup> The first user study was presented at the Refinement Workshop in 2019 (see Section A.2).

T. Runge, T. Thüm, L. Cleophas, I. Schaefer, and B. W. Watson [2019b]. “Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study”. In: *Formal Methods. FM 2019 International Workshops. Refine*. Vol. 12233. Lecture Notes in Computer Science. Springer, pp. 388–405. DOI: [10.1007/978-3-030-54997-8\\_25](https://doi.org/10.1007/978-3-030-54997-8_25)

The second user study was presented at the Formal Methods Teaching Workshop in 2021 (see Section A.3). We conducted the second user study to confirm our findings.

T. Runge, T. Bordis, T. Thüm, and I. Schaefer [2021]. “Teaching Correctness-by-Construction and Post-hoc Verification—The Online Experience”. In: *Formal Methods Teaching Workshop*. Vol. 13122. Lecture Notes in Computer Science. Springer, pp. 101–116. DOI: [10.1007/978-3-030-91550-6\\_8](https://doi.org/10.1007/978-3-030-91550-6_8)

### User Study Design

The design of the user studies is based on best practices given by Wohlin et al. [2012] with respect to quantitatively and qualitatively comparing two tools that are used for the same tasks.

<sup>5</sup><https://www.ueq-online.org/>

We consider the following three research questions to evaluate the approaches CbC and post-hoc verification (PhV) with the tools CORC and KEY.

**Q1:** What kind of errors do participants make with CbC and PhV?

**Q2:** What is the process of participants to create programs with CbC or PhV?

**Q3:** Which of the two approaches is preferred by users and why?

To be able to conduct the user studies, we need participants who understand the concepts of CbC with CORC, as well as post-hoc verification with KEY. Overall, three things are necessary to solve the tasks in the user study:

- Participants can specify methods with loop invariants and other auxiliary specification.
- Participants can verify methods with KEY. We will use KEY as automatic program verifier in the user studies. They do not have to know how to interact with KEY to prove programs, but they should understand what the reasons could be that a proof is not closed automatically by examining the open proof goals.
- Participants can construct programs with CORC.

The students of the *Software Quality 2* course at TU Braunschweig, a primary course in formal methods, are good candidates for the user studies. In the course, the students learn to specify and verify methods with KEY. They also learn correctness-by-construction and the CORC tool. Therefore, students who attend the whole *Software Quality 2* course are prepared for the user study. In the first user study, we had 10 participants, and 13 participants in the second user study.

Participants of our user study have to implement two algorithms, one with CbC and one with PhV. We had a time slot of 90 minutes for each user study. Therefore, the algorithms should have a suitable size to be implementable in that time slot. The first algorithm is *maximum element* that finds the maximum element in a given array of integers. The second algorithm is *modulo* that calculates the remainder of a dividend  $a$  and a divisor  $b$ . We prohibit the use of division and modulo operators in this task so that the participants have to use a loop to implement a *modulo* algorithm. Both algorithms have a similar size and cyclomatic complexity. The participants are arranged in two groups with the Latin square design. One group uses CbC for the first task, and PhV afterwards. The other group does the same tasks but uses the tools in reverse order. The tools are switched to address the possibility of learning effects resulting from the order of tool employment. For PhV, participants first fully implement the methods given the specification, and then use KEY subsequently to verify their correctness. In contrast, for CbC, participants use the graphical editor of CORC to construct the methods step-wise using the aforementioned refinement rules. To have the same starting point, we provide a pre-/postcondition specification for each algorithm.

To answer the research questions, we analyze defects in code and specification of the implemented program. Here, we are interested in the specific errors the participants make and if they are similar for both approaches. The programming procedure is analyzed by examining all intermediate snapshots of the participants. A snapshot is created every time a participant tries to verify the program. To get qualitative results regarding the comprehensibility and usability of the approaches, we take a user experience questionnaire (UEQ). The UEQ [Laugwitz et al. 2008] is a standardized

#Defects	PhV 1 <sup>st</sup>		CbC 1 <sup>st</sup>		PhV 2 <sup>nd</sup>		CbC 2 <sup>nd</sup>	
	Code	Spec.	Code	Spec.	Code	Spec.	Code	Spec.
<b>No Defects</b>	8	2	4	3	9	1	2	1
<b>Minor Defects</b>	1	7	3	4	4	10	4	5
<b>Major Defects</b>	1	0	1	0	0	0	0	0
<b>Incomplete</b>	0	1	2	3	0	2	7	7

Table 3.2.: Defects in code and specification of the final programs of participants [Runge et al. 2021]

questionnaire to compare two tools in terms of their usability. Six usability properties of a tool are measured: attractiveness, perspicuity, efficiency, dependability, stimulation, and novelty. To measure these aspects, a participant is asked to rate the tools with 26 items. Each item is a set of opposing adjectives describing the tool. The rating of the adjectives is a 7-point Likert-scale. We also ask open questions in a structured interview to get further insights how the participants liked the approaches.

## Results and Discussion

With the analysis of the developed programs and the answers of the questionnaire and the interview, we answer the research questions as follows [Runge et al. 2019b; Runge et al. 2021]. The generalizability of the results is limited due to the small number of participants.

**Q1.** *What kind of errors do participants make with CbC and PhV?* The first research question can be answered by analyzing the defects in the code and auxiliary specification (e.g., loop invariants) of participants. If the program cannot be corrected without changing five lines of code, we classify a program to have major defects. If fewer lines contain defects, we classify it to have minor defects. For defects in the auxiliary specification (i.e., loop invariant for both approaches and intermediate conditions for CbC), the same classification applies. Incomplete code or specification is classified separately. Table 3.2 shows the defects that participants have in their final programs in the first and second user study.

With PhV, programs were mostly implemented correctly by the respective participants. As the algorithms are small and not complex, we expected mostly correct results. In the first user study, one result has minor defects with PhV and one result has major defects with PhV in the code. In the second user study, we have four results with minor defects in the code. A typical defect is a loop guard that uses a wrong logical comparison operator (e.g., greater-than instead of greater-than-or-equal). The participants also initialized variables with incorrect values. Regarding the specification, nearly every result has minor defects, seven in the first user study, and ten in the second user study. A typical defect is a missing check that variables stay in a needed boundary. Some participants also missed specifying the loop variant.

With CbC in the first user study, we have three results with minor defects and one result with major defects in the code. In the second user study, four results have minor defects in the code. With CbC, the same defects as with PhV occur (i.e., wrong guards or wrong initialization of variables).

Only with CbC, we have incomplete coding results. The explanation given by the participants is that they have not the necessary knowledge to construct programs with CbC because they missed lectures or exercises. Regarding the specification with CbC, we have four minor defects in the first user study, and five minor defects in the second user study. We also have incomplete specification results. If the code was incomplete, the specification is likewise incomplete. With CbC, a typical defect is that the loop invariant does not hold initially or in the end.

The coding results are in favor of PhV in comparison to CbC. We argue that the familiar environment of writing Java code in a textual editor leads to fewer errors than using CoRC's graphical editor. Since we have more defects in the specification than in the code, writing correct specifications is more challenging for participants. Overall, we got worse results in the second study compared to the first user study as more incomplete results exist in the second user study. The reason is that more participants in the second user study missed lectures or exercises of the Software Quality course, and were therefore less prepared. Comparing the defects, the participants made similar errors with both approaches. For example, they introduced incorrect loop guards or initialized variables incorrectly. They also had the most problems with finding a sufficient loop invariant. This is in agreement with the results of Back [2009] that most of the time is spent to alter the loop invariant. With PhV, some participants forgot to specify a loop variant. This does not happen with CbC because of the dedicated field in the graphical editor for the loop variant. Otherwise, we did not identify real difference between CbC and PhV.

To answer the first research question, we observed the introduction of similar defects with both approaches. There are fewer defects with PhV than with CbC, but in our experience this is due to the familiarity of participants with writing code in textual editors. Furthermore, the participants do not have much expertise yet in constructing programs with CoRC. We assume that the defect rate decreases with a longer training period.

**Q2.** *What is the process of participants to create programs with CbC or PhV?* By analyzing all snapshots of intermediate programs in both user studies, we got insights in the programming process with CbC and PhV, respectively. With PhV, (1) the program was developed, (2) auxiliary specifications were added, and (3) the program was verified. Participants had mostly correct code in the first place, but the specification had defects. Interesting observations are that participants changed both the code and the specification if they could not verify the program. We saw that participants changed correct code several times when they could not verify the program, and surprisingly, the code remained correct. The participants did not notice that only the auxiliary specifications, such as loop invariants, were insufficient. The tool support was not sufficient, so that participants can pinpoint the defects in code or specification. The process to construct the programs was not monotonic after the first verification attempt. By monotonic, we mean that there is a clear process of specifying, constructing, and then verifying the program. The participants had rather a trial-and-error process to debug and verify the program.

With CbC, the participants interleaved writing code and specification, or they started with the specification. They even checked incomplete programs to get correctness guarantees for their applied refinements. However, even with CbC, participants changed a correct and verified part of the program if they could not prove correctness of the complete program. We cannot determine exactly what the participants needed to pinpoint the defects. The problem could be inadequate tool

support, inexperience with the tools, or inexperience with verification in general. With CbC, the participants had more verification attempts, but they changed the code less. We noticed that a sufficient specification helped to debug the program. For example, the participants found defects in the code after they introduced a correct loop invariant.

To answer the second research question, we found a non-monotonic trial-and-error process to construct programs with both approaches. The participants could not systematically fix bugs in the user studies. Nonetheless, the participants started the expected refinement process with CbC, but only up to the point where they could not solve a problem. We expect that further expertise with the tool leads to a more structured construction process.

**Q3.** *Which of the two approaches is preferred by users and why?* The third research question can be answered by analyzing the answers of the structured interview and the UEQ. The answers for the UEQ were summarized regarding the six measurements: attractiveness, perspicuity, efficiency, dependability, stimulation, and novelty. Overall, the average answers of the participants are in favor of CoRC. We measured a significant difference with the T-test for stimulation and novelty. For perspicuity, both tools have a negative mean value. KEY also has a negative value for novelty.

We clustered the answers of the open questions to analyze whether the participants had similar experiences. Some participants preferred post-hoc verification with KEY because of the familiar environment and syntax. CbC with CoRC has the advantage of the detailed feedback per statement as each statement is proved separately. The participants mentioned that the detailed feedback helps to locate problems in code or specification. In both user studies, the participants preferred CoRC over KEY. More participants would use CbC to construct correct programs and fixing defects. Considering that the participants made more errors with CoRC, the participants seem to factor in that they prefer correctness-by-construction over post-hoc verification. Surprisingly, no participant complained about the additional specification effort of writing intermediate conditions in CoRC, but they mentioned the changed and limited programming style due to the refinement rules.

To answer the third research question, the participants prefer CbC and CoRC over PhV and KEY due to the fine-grained feedback by checking individual statements. Some participants still prefer PhV because of the familiar environment.

## Related Work

Tool support for verification has already been evaluated in the literature, but PhV was not compared to CbC yet. We compare our user studies with related work that evaluated the use of verification tools. We also discuss related work on teaching formal methods because this is the foundation to have trained participants for user studies on CbC and program verification.

**User Studies with Verification Tools.** Related work for evaluating PhV or CbC are manifold, but before this work and to the best of our knowledge, there is no user study comparing both approaches qualitatively. Petiot et al. [2016] analyzed how developers interact with verification tools. They investigated how developers can be supported if they encounter open proof goals. One improvement is to categorize the occurred errors and calculate counter examples for each category. The calculation of counter examples is complementary to the CbC methodology. A defective Hoare triple could

be used as basis for counter examples to improve the user feedback. Johnson et al. [2013] interviewed developers about their use of static analysis tools. The most important point mentioned is that developers need good error reporting. Hentschel et al. [2016b] evaluated how formal methods supports code reviews. Their symbolic execution debugger (SED) [Hentschel 2016; Hentschel et al. 2016a] helped to locate defects in already existing programs. The interaction of developers with the program verifier KEY during the verification of programs is also analyzed [Beckert et al. 2014a; Beckert et al. 2014b]. The authors identified that developers need much effort to understand the current proof state so that they can interact with the tool properly. In comparison to the related works, we focus on the usability of the tools during program constructions in our user studies. The developers used CORC and KEY as automatic program verification tools. Therefore, we excluded the interaction with program verifiers in our user studies. We analyzed the programming procedure of the developers and how they assess the tools to detect defects in the code or specification.

Back [2009] evaluated the CbC tool SOCOS. They state that correctness-by-construction needs well-developed tools to support users during the refinement process. The majority of the time, developers refine the invariant to a correct solution. The correct invariant is found by iterative refinement of a partial (and possibly wrong) invariant. In our user studies, we have observed the same behavior that loop invariants are altered several times. With correct and sufficient loop invariants, participants usually created a correct and verified program.

**Teaching Formal Methods.** Teaching formal methods has been researched in related work [Cataño 2019; Creuse et al. 2019; Divasón and Romero 2019; Liu et al. 2009], where the authors discuss teaching experiences and evaluate how students learn with different teaching strategies and the support of tools. Liu et al. [2009] discovered that a mix of pen-and-paper and tool-supported exercises are beneficial for students to learn the material. By writing on paper, the students consolidate what they have learned. With tools, larger assignments can be completed because the students are supported in their tasks. Students also increase their productivity with the tools. Being able to solve problems in tools and receive positive feedback when they verify a program also increases student interest. Creuse et al. [2019] state that teaching by example is valuable to get started with formal methods. Practical examples facilitate the entry to formal methods. Cataño [2019] detect that students need immediate and understandable feedback during their specification or verification process. Students want to understand whether problems in the process occur. We differ from this related work [Cataño 2019; Creuse et al. 2019; Divasón and Romero 2019; Liu et al. 2009] because we do not focus on teaching. It is a mandatory prerequisite for our user studies that students know the tools. Therefore, we teach PhV with KEY and CbC with CORC in the Software Quality 2 course. The course is guided by best practices from related work. CORC (with its graphical interface) is well suited for teaching CbC, since program development and verification feedback are presented in detail.

## Conclusion

The goal of the user studies [Runge et al. 2019b; Runge et al. 2021] is to evaluate the usability of CbC through CORC to answer *Research Question RQ<sub>2</sub> – Usability of Tool-Supported Correctness-by-Construction*. In the user studies, the participants wrote mostly correct code with PhV. We have more defects with CbC. In case of writing auxiliary specification, we have about the same number of cor-



rect results for both approaches. From these results, we conclude that the participants need more time to gain expertise with CbC and CORC to reduce the defect rate in the code. They are familiar with textual editors to write small algorithms, but are not used to CORC’s graphical editor. The exact errors made are similar for both approaches; the participants wrote guards with wrong logical comparison operator or too weak loop invariants. In the interviews, the participants stated that they prefer the structured programming approach of CbC, and they appreciated the detailed feedback of CbC and CORC. The verification of the individual steps helps to identify the source of the error. The familiar coding environment was the main factor in favor of post-hoc verification. Therefore, we consider that CbC can be an alternative to PhV. As some participants made more errors with CbC and also criticized the limited flexibility of the CbC programming approach, this led us to **Challenge 2** and our third research question.

### 3.3. Alternative Correct-by-Construction Program Development Approaches

As **Challenge 2**, we identified that correctness-by-construction as proposed by Kourie and Watson [2012] has a rigid program construction process. In our user studies, we received feedback from the participants that the construction rules restrict developers in their programming tasks. Therefore, we want to investigate alternative correct-by-construction program development approaches that address the limited flexibility of correctness-by-construction by Kourie and Watson [2012].

In this section, we develop two alternative correctness-by-construction approaches and compare correctness-by-construction by Kourie and Watson [2012] with these approaches. We want to answer the research question RQ3: *What alternative correct-by-construction program development approaches exist, and how do they compare to rule-based correctness-by-construction?* CBC-BLOCK, the first CbC development approach aims to relax the strict guideline of CbC. CBC-BLOCK introduces new refinement rules that condense any number of refinement rule applications into one construction step. TRAITCBC, the second CbC development approach is composition-based and relies on traits [Ducasse et al. 2006] to construct correct and reusable programs. In TRAITCBC, small units of code (i.e., methods in traits) are constructed and verified. Then, these units are composed to larger programs in a correct-by-construction manner. Therefore, TRAITCBC does not need refinement rules to construct programs. In this section, CbC by Kourie and Watson [2012] is referred as classic CbC to be precise which correct-by-construction program development approach is meant. TRAITCBC was presented at the International Conference on Formal Techniques for Distributed Objects, Components, and Systems (see Section A.4). Our work was awarded the Best Paper Award.

T. Runge, A. Potanin, T. Thüm, and I. Schaefer [2022b]. “Traits: Correctness-by-Construction for Free”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Vol. 13273. Lecture Notes in Computer Science. Springer, pp. 131–150. DOI: [10.1007/978-3-031-08679-3\\_9](https://doi.org/10.1007/978-3-031-08679-3_9)

As an extended publication, we introduced CBC-BLOCK in the special issue of the Journal on Logical Methods in Computer Science (see Section A.5).

T. Runge, T. Bordis, A. Potanin, T. Thüm, and I. Schaefer [2023]. “Flexible Correct-by-Construction Programming”. *Logical Methods in Computer Science*

## CbC-BLOCK

CbC-BLOCK extends CbC by adding two refinement rules that allow to introduce a block of program statements instead of just one as given by the classic CbC refinement rules. These new rules allow a simpler way to construct a program as the application of several classic CbC refinement rules can be condensed by introducing one block of code. Therefore, the flexibility to construct programs is improved. The rigid program development approach of classic CbC is loosened while retaining the benefits of a structured program construction approach (e.g., a guideline to construct programs, and correctness guarantees for each refinement step).

The first rule (block-introduction) introduces a block statement by refining one abstract statement. The block statement is similar to an abstract statement, but it is explicitly specified with a pre-/postcondition contract (i.e., a block contract [Ahrendt et al. 2016]). To guarantee that this refinement retains the correctness of the program under development, we have to check three parts. First, the precondition of the refined abstract statement must imply the precondition of the block. We have to ensure that the pre-state of the block is satisfied to be able to execute the block. Second, the postcondition of the block must imply the postcondition of the refined abstract statement. Third, the block must satisfy its own contract. As the block is still abstract, it must be refined first to prove this third proof obligation. [Definition 3.1](#) shows the block-introduction rule formally.

### Definition 3.1: Block-Introduction

Let  $\{P\}S\{Q\}$  be a Hoare Triple with precondition  $P$ , abstract statement  $S$ , and postcondition  $Q$ . Then, Hoare triple  $\{P\}S\{Q\}$  is **refinable** to  $\{P'\}$  Block B  $\{Q'\}$  iff  $P$  implies  $P'$  and  $Q'$  implies  $Q$  and  $\{P'\}$  Block B  $\{Q'\}$  holds, where  $P'$  and  $Q'$  are pre- and postcondition of Block B.

The block statement can be refined in two ways. First, any already existing CbC refinement rule can be applied, and the correctness is guaranteed as if an abstract statement is refined with one of the refinement rules. Second, the block can be refined with the new block-instantiation rule to any sequence of concrete program statements. The introduced block of code can have any size from simple code to a complete method implementation. Thus, the application of the block-instantiation rule can condense the application of several CbC refinement rules. The side conditions of the block-instantiation rule guarantee that the introduced sequence of program statements fulfills the pre-/postcondition contract of the block. To prove fulfillment of the contract, program verification is used. Similar to a method call, we prove that the dynamic logic formula  $P \rightarrow \langle \text{statement}; \dots \rangle Q$  is fulfilled. The formula denotes that, if the precondition is assumed, and the code is executed, the code terminates, and the program is in a state that satisfies the postcondition. [Definition 3.2](#) shows the block-instantiation rule formally.



**Definition 3.2: Block-Instantiation**

Let  $\{P\}S\{Q\}$  be a Hoare Triple with precondition  $P$ , block statement Block B, and postcondition  $Q$ . Then, Hoare triple  $\{P\}$  Block B  $\{Q\}$  is **refinable** to  $\{P\} < \text{statement}; \dots > \{Q\}$  iff  $P \rightarrow < \text{statement}; \dots > Q$ , where  $< \text{statement}; \dots >$  is any sequence of concrete program statements possibly containing further blocks.

Classic CbC already has a method call rule that can be used to refine an abstract statement to a call of any auxiliary code. With the block refinement rule, we can introduce any code without the overhead of creating new methods. The main advantage of a block is that local variables of the surrounding context can be altered. With CbC, we always specify the complete program state in any condition (e.g., pre-/postcondition) to exclude the possibility of unwanted side effects in the block that could be missed. A method call cannot alter local variables of the calling context. The block-instantiation rule also enforces that the code sequence is verified against the pre-/postcondition specification, then its contract can be assumed for the verification of the surrounding context. For a called method, it is just assumed that the method fulfills its contract. This has to be proved separately.

**Example 3.2.** *To give an example, we implement a `maxElement` algorithm. The algorithm finds the largest element in a list of integers. The list offers a `get`-method to access an element of the list at a specific position. A `contains`-method checks that the result is a member of the list. In [Listing 3.1](#), we show a starting point of the program where some code is already developed using CbC refinement rules. The program is specified with a pre-/postcondition contract. The precondition states that the list contains at least one element. The postcondition states that the largest element in the list is returned. The program starts with declaring two local variables in lines 6 and 7. The variable `i` stores the current largest element, and variable `j` is used to iterate through the list. In lines 11–15, a block is introduced with the block-introduction rule. The block is specified with a pre-/postcondition contract. The precondition states the initial values of the local variables, and the postcondition states that `i` the largest element. As the block is also surrounded with intermediate conditions in lines 9 and 17 (these were already introduced with other CbC rules), we can check if the side conditions of the block-introduction rule are fulfilled. We have to show that the condition before the block implies the precondition of the block, and that the postcondition of the block implies the condition after the block. Both implications are valid in the example.*

```

1 /*@ requires list.size() > 0;
2   @ ensures list.contains(\result) && (\forall int q;
3     @   q >= 0 && q < list.size(); \result >= list.get(q));
4   @*/
5 public int maxElement(List list) {
6     int i = list.get(0);
7     int j = 1;
8
9     //@ Intm: list.size() > 0 && i == list.get(0) && j == 1;
10
11    /*@ requires list.size() > 0 && i == list.get(0) && j == 1;
12      @ ensures list.contains(i) && (\forall int q;
13        @   q >= 0 && q < list.size(); i >= list.get(q));
14      @*/

```

---

```

15     { \Block B1; }
16
17     //@ Intm: list.contains(i) && (\forall int q;
18     //@   q >= 0 && q < list.size(); i >= list.get(q));
19
20     return i;
21 }

```

---

Listing 3.1.: Initial program of maxElement

In Listing 3.2, the block B1 is concretized with the block-instantiation rule. The instantiated block must fulfill its contract. The implementation of the block in lines 8–17 introduces a loop to iterate through the list and to update the variable `i` if a larger element is found. The loop is also specified with a loop invariant and variant. The invariant states that between the indices `0` and `j` the current largest element is stored in `i`. We have to show that the instantiation satisfies the block contract. Therefore, we prove that if the precondition is satisfied and after executing the block, the postcondition must be fulfilled. This can be proved with established program verifier. When this obligation is proved, the program is fully refined and verified. This instantiation condenses the application of several CbC refinement rules, the repetition rule to create the loop, a composition rule, a selection rule to check for a larger element, and two assignment rules. Therefore, the block rules provide a flexible way to complete the program without relying on many classic CbC refinement rules.

---

```

1Block B1;
2
3/*@ requires list.size() > 0;
4 @ ensures list.contains(i) && (\forall int q;
5 @   q >= 0 && q < list.size(); i >= list.get(q));
6 @*/
7 {
8   //@ loop_invariant list.contains(i)
9   //@   && j > 0 && j <= list.size() &&
10  //@   (\forall int q; q >= 0 && q < j; i >= list.get(q));
11  //@ decreases list.size() - j;
12   while (j < list.size()) {
13     if (list.get(j) > i) {
14       i = list.get(j);
15     }
16     j = j + 1;
17   }
18 }

```

---

Listing 3.2.: Refinement of block B1

CbC-BLOCK is implemented by extending CORC to support the refinement rules of CbC-BLOCK. For the verification of an introduced block (Definition 3.1), we use KEY to prove the side conditions that the pre-/postcondition are suitable. For the verification of the block-instantiation (Definition 3.2), we implemented a generator that transforms a block to a method in order to use KEY

for method verification. Thus, we can reuse an existing feature of KEY. CORC's usability was substantiated with two user studies in Section 3.2. We conducted an additional user study for the CBC-BLOCK extension [Runge et al. 2023]. We compared CBC-BLOCK with classic CbC by (dis)allowing the use of the block rules. The structure is otherwise the same as the previous user studies. In the additional user study for CBC-BLOCK, the participants appreciated the grouping of statements to one block of code, and the freedom to not be bound to the classic CbC refinement rules. All participants considered the introduction of the block rules useful.

## TRAITCBC

TRAITCBC is based on *traits* [Ducasse et al. 2006] that constitute a flexible language construct for modular code reuse. A trait is independent of any class hierarchy and consists of *concrete* or *abstract* methods. Traits can be composed to larger traits or classes containing all methods of all composed traits. During composition, the compatibility of methods is checked. This means, the specification of a concrete method has to satisfy the specification of an abstract method with the same signature (see Liskov substitution principle [Liskov and Wing 1994]).

TRAITCBC uses traits and trait composition to enable an incremental program development approach. The development approach is as follows: All abstract or concrete methods that are introduced in TRAITCBC are specified with a pre-/postcondition contract. A developer implements a first trait with a concrete method *m1*. This method *m1* contains *holes*. A hole is an abstract method (e.g., an abstract method *m2*) that is called in method *m1*. The abstract method *m2* is also part of the first trait. With this information, we can verify that method *m1* fulfills its contract under the assumption that method *m2* fulfills its contract. In the next step, the abstract method *m2* is implemented in a second trait, where the implementation of method *m2* can contain further abstract methods. After the correctness of method *m2* is also proved, the two traits are composed. We check that the contract of the concrete method *m2* in the second trait fulfills the contract of the abstract method *m2* in the first trait. This incremental TRAITCBC approach stops when all abstract methods are implemented, and all traits are composed. TRAITCBC is based on post-hoc verification [Ahrendt et al. 2016]. Therefore, the same programs can be verified with TRAITCBC as with post-hoc verification, but in addition, TRAITCBC introduces an explicit program construction approach. It utilizes the flexibility of traits, which is beneficial for scenarios as incremental development [Damiani et al. 2014] and the development of software product lines [Clements and Northrop 2002; Bettini et al. 2010].

**Example 3.3.** We implement the `maxElement` algorithm in a slightly different way than before to better fit the TRAITCBC approach. In TRAITCBC, programs are organized in reusable traits with concrete and abstract methods. Therefore, we strive for methods that are verified once, but called several times in the program. The first implementation step is shown in Listing 3.3. In trait `MaxETrait1`, an abstract method `maxElement` is specified with a pre-/postcondition contract. The precondition excludes empty lists, and the postcondition states that the result is the largest element in the list.

---

```
1 trait MaxETrait1 {
2 /*@ requires list.size() > 0;
3  @ ensures list.contains(\result) &
4  @   (\forall int n: list.contains(n) ==> \result >= n);
5  @*/
```

---

```

6  abstract int maxElement(List list);
7}

```

---

Listing 3.3.: Initial trait for maxElement

As second step, we implement the method in trait MaxETrait2 shown in Listing 3.4. The implementation of maxElement is as follows. If there is only one element in the list, we found the maximum. If the first element is larger than the other elements in the tail of the list, we return this first element. Otherwise, the maximum is searched in the tail of the list. The method implementation calls two abstract methods. The abstract method accessHead accesses the first element of the list. The abstract method maxTail finds the maximum element in the tail of the list, so it recursively searches for the largest element.

---

```

1trait MaxETrait2 {
2/*@ requires list.size() > 0;
3  @ ensures list.contains(\result) &
4  @   (\forall int n: list.contains(n) ==> \result >= n);
5  @*/
6  int maxElement(List list) {
7    if (list.size() == 1) {return accessHead(list);}
8    else if (accessHead(list) >= maxTail(list))
9      {return accessHead(list);}
10   else {return maxTail(list);}
11 }
12
13/*@ requires list.size() > 0;
14  @ ensures \result == list.element();
15  @*/
16 abstract int accessHead(List list);
17
18/*@ requires list.size() > 1;
19  @ ensures list.tail().contains(\result) &
20  @   (\forall int n: list.tail().contains(n) ==> \result >= n)
21  ;
22  @*/
23 abstract int maxTail(List list);
24}

```

---

Listing 3.4.: Implementation of maxElement with auxiliary methods

In this trait, we can verify the correct implementation of the method maxElement under the assumptions that both abstract methods will be correctly implemented in subsequent construction steps. In TRAITCBC, we verify the correctness of maxElement directly, and then proceed to compose both traits MaxETrait1 and MaxETrait2. When composing the traits, we prove that the specification of the concrete method maxElement fulfills the specification of the abstract method. In concrete, we prove that:

MaxETrait1.maxElement(..).pre ==> MaxETrait2.maxElement(..).pre as well as:

MaxETrait2.maxElement(..).post ==> MaxETrait1.maxElement(..).post.

By composing two correct traits, we know that the resulting trait is correct by construction. In this example, the

specifications are the same, so proving the composition is trivial, but this is not generally the case. The next steps would be to implement the remaining abstract methods and compose the traits to finalize the implementation.

TRAITCBC requires two properties to guarantee the correctness of programs under development. First, we are using program verification to prove the correctness of concrete methods. Each concrete method in a trait is directly verified with the information that is present in its trait. Therefore, all called methods of a concrete method must be part of the trait (either abstract or concrete). Second, for trait composition, we use verifiers to check the compatibility of methods: a concrete method must have a weaker precondition and a stronger postcondition to satisfy the contract of the abstract method with the same signature. [Definition 3.3](#) shows the method-composition rule formally. When composing two methods, we have four cases. (1) If a concrete method is composed with an abstract method, the precondition of the abstract method has to imply the precondition of the concrete method, and the postcondition of the concrete method has to imply the postcondition of the abstract method (see Liskov substitution principle [[Liskov and Wing 1994](#)]). (2) The second case is symmetric to the first case. (3) If two abstract methods are composed, the contract of one method has to imply the contract of the other method. This is needed so that a concrete method can satisfy the contracts of both abstract methods. (4) The fourth case swaps the implication direction of the contracts of the two abstract methods. The composition of two concrete methods is correctly left undefined. The user has to solve this case manually by choosing one implementation.

#### Definition 3.3: Method Composition

Let  $M_1$  and  $M_2$  be two methods. A method has the signature `method C m(C1 x1 ... Cn xn) e;` with a name  $m$ , a return type  $C$ , a list of parameters  $C_i x_i$ , and a body  $e$ . An abstract method has a similar signature `method C m(C1 x1 ... Cn xn);` without the body  $e$ . A method is specified with a contract  $S$ , where the pre- and postcondition are accessed with the functions  $Pre(S)$  and  $Post(S)$ .  $MH$  is the method header without the body.

Then  $M_1 + M_2 = M$  is the composition of these methods with resulting method  $M$ .

- `S method C m(C1 x1 ... Cn xn) e; + S' method C m(C1 ... Cn -);`  
 $=$  `S method C m(C1 x1 ... Cn xn) e;`  
*if  $Pre(S')$  implies  $Pre(S)$  and  $Post(S)$  implies  $Post(S')$*
- `MH1; + MH2 e; = MH2 e; + MH1;`
- `S method C m(C1 x1 ... Cn xn); + S' method C m(C1 ... Cn -);`  
 $=$  `S method C m(C1 x1 ... Cn xn);`  
*if  $Pre(S')$  implies  $Pre(S)$  and  $Post(S)$  implies  $Post(S')$*
- `S method C m(C1 x1 ... Cn xn); + S' method C m(C1 ... Cn -);`  
 $=$  `S' method C m(C1 x1 ... Cn xn);`  
*if ( $Pre(S)$  implies  $Pre(S')$  and  $Post(S')$  implies  $Post(S)$ )*  
*and not ( $Pre(S')$  implies  $Pre(S)$  and  $Post(S)$  implies  $Post(S')$ )*

In our work [[Runge et al. 2022b](#); [Runge et al. 2023](#)], we established the syntax, type system, reduction rules, and trait flattening semantics for TRAITCBC. We formalized a program development approach using pre-/postcondition contracts and method calls instead of refinement rules and abstract statements as in classic CbC. We proved soundness of the trait composition process and that TRAITCBC enables an incremental construction approach. An informal version of the theorem that

	Classic CbC	CbC-BLOCK	TRAITCbC
Language Constructs	Adds refinement rules to a programming language. Needs specification language.	Adds refinement rules to a programming language. Has new refinement rules to introduce a specified block of statements. Needs specification language.	Programming language with traits. Needs specification language.
Tool support	Needs specialized tool support.	Needs specialized tool support. Implemented as extension of CoRC.	Relies on post-hoc verification tools.
Development/Verification	Specific refinement rules guarantee the correctness of each refinement step.	Specific refinement rules guarantee the correctness of each refinement step. Refinements can be condensed with the block rules.	Flexible construction by composition of traits. Each method is specified so that each constructed method can directly be verified.

Table 3.4.: Comparison of classic CbC with CbC-BLOCK and TRAITCbC [Runge et al. 2023]

the TRAITCbC approach constructs correct programs is presented in [Theorem 3.1](#). Starting with a set of verified traits, the composition of these traits is a verified program. The formal theorem is established and proved in the paper [Runge et al. 2023].

#### Theorem 3.1: Sound CbC Process

Starting with a set of verified traits  $t_0 \dots t_n$ , we can write  $C = t_0 + \dots + t_n$  as our TRAITCbC approach; where  $t_0 + t_1$  is the application of the first construction step (trait composition), and  $t_0 + t_1 + t_2$  is the application of the second construction step, etc.

Then we get a verified program  $C$  correct by construction.

The implementation of TRAITCbC is based on Java and JML. In this setting, a trait is an interface with default implementations. We provide an editor to state which traits should be composed for the incremental construction of complete programs. The editor also starts the verification process. For the verification of methods and method composition, existing program verifier can be used. Our implementation of TRAITCbC uses KEY [Ahrendt et al. 2016] for Java/JML program verification. TRAITCbC was evaluated with a feasibility study [Runge et al. 2022b], where we showed that case studies can be implemented at varying levels of granularity, from small (auxiliary) methods to complex ones.

## The CbC Program Development Approaches in Comparison

We compare CbC by Kourie and Watson [2012] (classic CbC) with CbC-BLOCK and TRAITCbC regarding the language constructs to write programs, tool support, the program development approach, and verification process to discuss the advantages and drawbacks of the approaches. The comparison is summarized in [Table 3.4](#) [Runge et al. 2023].

*Language Constructs.* All approaches are based on a programming and specification language to write and specify programs. The refinement rules of classic CbC and CbC-BLOCK are external to a programming language. By external, we mean that refinement rules are not part of the programming language; they are a program transformation concept to refine an abstract statement to a concrete implementation. For TRAITCbC, traits must be present in the language to allow



the composition-based construction approach (e.g., Java has interfaces with default implementations that are a good approximation of traits). The advantage of `TRAITCbC` is that no external refinement rules are necessary to develop programs.

*Tool Support.* For classic CbC, the `CORC` tool was developed (see [Section 3.1](#)). `CORC` was extended in this contribution to also support the refinement rules of `CbC-BLOCK` [[Runge et al. 2023](#)]. The feasibility of `CORC` was evaluated with several case studies that could be implemented in `CORC`. We measured a similar verification effort as post-hoc verification, but with the advantage that more programs could be verified with `CORC` [[Bordis et al. 2022a](#)]. The usability of `CORC` and the extended version for `CbC-BLOCK` are shown with user studies [[Runge et al. 2019b](#); [Runge et al. 2021](#); [Runge et al. 2023](#)]. The participants appreciated the general fine-grained construction that helps to detect errors, but also the possibility to condense refinement steps with the block rules. `TRAITCbC` is implemented with an editor to state the trait composition and start the verification process with `KEY`. We showed the feasibility by implementing case studies in varying levels of granularity.

Therefore, all approaches are tool-supported, and it is feasible to construct programs with any of the approaches. For `TRAITCbC`, a specialized editor is beneficial, but general program verification tools are sufficient. Classic CbC and `CbC-BLOCK` need specialized tools to support the rule-based refinement approach.

*Program Development and Verification.* Classic CbC has a strict guideline to construct programs with refinement rules. `CbC-BLOCK` relaxes this guideline by introducing blocks of code with the new refinement rules. The `CbC-BLOCK` refinement rules can save the application of several other refinement rules. In both approaches, checking the side conditions of the applied refinement rule guarantees the correctness of the program under construction. `TRAITCbC` also relaxes the rigid guideline as methods of any size can be developed in the traits. Nonetheless, `TRAITCbC` ensures that the implemented methods are directly verified regarding their specification. Afterwards, the traits and their methods are composed to correct software. `TRAITCbC` endorses to construct code in fine-grained steps that are more amenable for verification than single complex methods. These methods are also easily reusable in other contexts because they are implemented in traits that can be composed with other traits as needed.

All three approaches, provide an incremental program construction approach that ensures the correctness of the program in each step. `CbC-BLOCK` offers more freedom compared to classic CbC. `TRAITCbC` also provides a flexible way to develop programs, even without relying on refinement rules.

## Related Work

We discussed related work on correctness-by-construction and program verification before. Since `CbC-BLOCK` is an extension of CbC by Kourie and Watson [[2012](#)], the same differentiation as before applies. `CbC-BLOCK` utilizes block contracts [[Ahrendt et al. 2016](#)] that specify the behavior of a Java block similar to a method [[Meyer 1992](#); [Leino 1995](#)]. To establish a CbC refinement process, we introduced a refinement rule that refines an abstract statement to a specified block, and a refinement rule that instantiates a block.

The main difference between related CbC approaches and `TRAITCbC` is that `TRAITCbC` is composition-based, where atomic units of code are correctly composed to complete programs. No

refinement rules are necessary to construct programs. In the following, we discuss related work on the concept of traits in programming and how it relates to formal verification.

For clean design and reuse, traits are introduced in many languages, for example, in Smalltalk [Ducasse et al. 2006], Java [Bono et al. 2014] by utilizing default methods in interfaces, and other Java-like languages [Bettini et al. 2013; Liquori and Spiwack 2008; Smith and Drossopoulou 2005]. Other languages also use the term trait, but traits in Scala are mixins [Flatt et al. 1998], and traits in Rust are type classes [Sozeau and Oury 2008]. Traits in Smalltalk [Ducasse et al. 2006] are stateful. They have private fields to reduce the amount of required get-methods to access state. In comparison to Smalltalk, traits in our language are stateless. We rely on abstract state operations to represent state. A class is instantiable if its only abstract methods are valid get-methods. The get-methods and other methods can also be renamed [Bettini et al. 2013; Reppy and Turon 2006], allowing flexible reuse. To verify traits, Damiani et al. [2014] established a post-hoc verification approach for the trait language TraitRecordJ [Bettini et al. 2013]. Damiani et al. [2014] proposed a modular and incremental verification process for contract-based specified methods in traits. None of the related work formulated a CbC refinement process to create correct programs based on traits.

## Conclusion

The goal of this contribution is to address the rigid program construction of classic CbC (**Challenge 2**). We proposed two related CbC approaches, CBC-BLOCK and TRAITCBC, that have a relaxed guideline to construct correct programs by construction. CBC-BLOCK is a rule-based CbC approach with refinement rules that allows a more flexible program construction than classic CbC. The block refinement rules allow to condense the application of any number of classic CbC refinement rules. CBC-BLOCK is implemented in CORC and evaluated with a user study. The participants of the user study appreciated the increased flexibility enabled by the block rules. TRAITCBC is a new approach that uses composition to construct programs. TRAITCBC is therefore an alternative CbC approach with a program development process that does not need refinement rules. Rather, small program parts (auxiliary methods) are developed and composed with regard to their specification. TRAITCBC is implemented with an editor for trait composition and a standard post-hoc program verifier. We showed feasibility of this composition-based program development approach by implementing several case studies. We addressed *Research Question RQ3 – Alternatives to Rule-based Correctness-by-Construction* with the comparison and discussion of classic CbC, CBC-BLOCK, and TRAITCBC regarding their characteristics.

## 3.4. A Uniform Correct-by-Construction Program Development Approach for Functional Correctness and Security

In this section, we investigate how to construct functionally correct and secure programs. It is beneficial if both properties are ensured at the same time, however, they are normally checked one after the other. This can cause problems, for example, if the functional correctness of a program is already verified, but an information flow analysis detects a problem that needs a change in the



program, we have to reverify the changed program because the changes could affect the functional correctness. By ensuring both properties in a uniform CbC process, this problem can be mitigated.

The structured CbC approach is already advantageous for functional correctness [Watson et al. 2016]. With our third contribution [Runge et al. 2020; Runge et al. 2022c; Runge et al. 2022a], we want to exploit the structured development and reasoning process of CbC for secure programs. Thus, we want to answer the fourth research question RQ4: *How can we support the development of programs with correctness-by-construction that are functionally correct and satisfy information flow security?* To answer this question, we first create SIFO, a new information flow security type system for an object-oriented language [Runge et al. 2022c]. This type system type-checks whether developed programs fulfill the defined information flow policy. In general, the type system ensures that an attacker is never able to deduce private data by observing public data. We decided to use SIFO as basis for our uniform CbC process because SIFO utilizes immutability and uniqueness properties of objects for the detection of information leaks. By knowing that specific objects are only accessible through a unique reference or that objects are immutable, we can make precise statements about potential information flow to not conservatively reject secure programs. Already existing type systems for Java-like languages [Sabelfeld and Myers 2003; Myers 1999; Banerjee and Naumann 2002] do not have this kind of mutation or alias analysis and therefore pessimistically reject secure programs. SIFO was presented as article in the Journal ACM Transactions on Programming Languages and Systems (TOPLAS) (see Section A.6).

T. Runge, M. Servetto, A. Potanin, and I. Schaefer [2022c]. “Immutability and Encapsulation for Sound OO Information Flow Control”. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. DOI: [10.1145/3573270](https://doi.org/10.1145/3573270)

In a second step, we transform the typing rules into refinement rules to establish a correct-by-construction program development approach for information flow security. By integrating the refinement rules in the tool CORC, we support CbC for functional correctness and information flow security. A previous version of CbC for secure information flow for an imperative language was presented at the International Conference of Formal Methods in Software Engineering (see Section A.7).

T. Runge, A. Knüppel, T. Thüm, and I. Schaefer [2020]. “Lattice-Based Information Flow Control-by-Construction for Security-by-Design”. In: *FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering*. ACM, pp. 44–54. DOI: [10.1145/3372020.3391565](https://doi.org/10.1145/3372020.3391565)

The uniform CbC approach for an object-oriented language that is based on SIFO was then presented at the International Conference on Software Engineering and Formal Methods (see Section A.8).

T. Runge, A. Kittelmann, M. Servetto, A. Potanin, and I. Schaefer [2022a]. “Information Flow Control-by-Construction for an Object-Oriented Language”. In: *International Conference on Software Engineering and Formal Methods*. Vol. 13550. Lecture Notes in Computer Science. Springer, pp. 209–226. DOI: [10.1007/978-3-031-17108-6\\_13](https://doi.org/10.1007/978-3-031-17108-6_13)

## SIFO

SIFO is an object-oriented and expression-based language that is similar to Featherweight Java [Igarashi et al. 2001]. In SIFO, each reference and expression is associated with a type  $T$ . The type  $T$  is composed of a security level  $s$ , a type modifier  $mdf$ , and a class name  $C$ . As already described in the background, the security levels are arranged in a lattice (see Definition 2.2). We allow the information flow from lower to higher security levels with promotion typing rules. The promotion rules allow in secure cases that an expression with a lower security level is treated as an expression with a higher security level. The opposite flow of information is prohibited, except the developer uses a declassify-expression. Here, an expression with a higher security level is treated as an expression with a lower security level.

The type modifier  $mdf$  can be *imm*, *mut*, *capsule*, and *read* [Giannini et al. 2019]. An *imm* reference points to an immutable object. An immutable object cannot be updated after creation, but it can be aliased. A *mut* reference points to a mutable object. A mutable object can be updated and aliased. A *capsule* reference points to a mutable object, but that object and the mutable objects in its reachable object graph can only be accessed from this reference. This capsule reference can be used only once to assign the object to any other reference, then the property of the unique reference is lost. A *read* reference points to a mutable or immutable object, but that object cannot be mutated or aliased from the read reference. There is no immutability guarantee that the object is not accessible by other references. With these modifiers, we can make precise statements about the information flow in a program. We utilize immutability or uniqueness properties of the objects to estimate if, for example, a public object can be updated with private information, or an unwanted public alias to a private object can exist.

**Example 3.4.** *To give concrete examples for the reasoning with SIFO, we present the implementation of a class `Card` with fields `low imm int number` and `high mut Balance blc` in Listing 3.5.*

---

```
1 class Card{low imm int number; high mut Balance blc;}
2 class Balance{low imm int blc;}
```

---

Listing 3.5.: Class declarations

In Listing 3.6, we show allowed and prohibited assignments with immutable objects, as information flow reasoning is the easiest with these immutable objects. When accessing a field of an object, the security level of the returned value is calculated by the least upper bound of the accessed field security level and the receiver security level. Therefore, the value of the field `blc` of `Card` object `c` is high and can be assigned to a high reference in Line 4. In Line 5, the assignment of a high value to a low reference is not allowed because this assignment leaks confidential information when an attacker reads the low `b` reference. In Line 6, the low immutable expressions is assigned to the field `c.blc.blc`. The field `c.blc.blc` is typed by *imm*. Immutability of a field guarantees that the assigned object cannot be mutated. The `Balance` object `c.blc` is mutable, therefore, a new value can be assigned to the field `c.blc.blc` as long as the assigned value is immutable as declared by the field. The field `c.blc.blc` has a high security level because the chained expression contains the high field `blc` of class `Card`. Therefore, a promotion of the expression `c.number` to high is needed to be securely assignable to the field. The promotion is secure because the expression is immutable and cannot be mutated to introduce new confidential information. In Line 7, the opposite update of a low field with a high `int` is prohibited because this is a direct flow from a higher to a lower security level which violates the security policy.

---

```

3 low mut Card c = new low Card(); //an existing Card reference
4 high mut Balance blc = c.blc; //correct access of high blc
5 low imm int b = c.blc.blc; //wrong high assigned to low
6 c.blc.blc = c.number; //correct update with promoted imm int
7 c.number = highInt; //wrong, high int assigned to low c.number

```

---

Listing 3.6.: Examples with immutable objects

In Listing 3.7, we show secure and insecure updates with mutable objects. We enforce a strict separation of mutable objects with different security levels. With that separation, we prohibit that an update of an object through a higher reference is read by lower references afterwards. In Line 8, a new Balance object is initialized as low. This new Balance object itself is not confidential, but we want to make it confidential by assigning it to c.blc in Line 9. The field blc of class Card is high. Therefore, this assignment is prohibited. If Line 9 would be accepted, we could access c.blc.blc to update the balance integer with a private value. That change is readable by the still existing newBlc reference, and this is a leak of the introduced private information. In Line 10 and 11, an assignment without aliasing is shown using the capsule type modifier. The referenced low capsBlc object is promoted to a high security level and assigned in Line 11. This assignment is secure because the capsule reference is no longer accessible after its first use. Thus, it is not possible to read private information via the capsBlc reference.

---

```

8 low mut Balance newBlc = new low Balance(0); //ok
9 c.blc = newBlc; //wrong, mutable secret shared as low and high
10 low capsule Balance capsBlc = new low Balance(0); //ok
11 c.blc = capsBlc; //ok, no alias introduced

```

---

Listing 3.7.: Examples with mutable and encapsulated objects

In our work [Runge et al. 2022c], we formalized SIFO by presenting, syntax, typing rules, and reduction rules. To ensure correctness of the type system, we prove *noninterference*: private data should never influence public data. With this property, we ensure that an attacker can never deduce private information by observing data with lower security levels. We implemented the type system and evaluated feasibility by constructing several case studies. We constructed five case studies where we discovered that we can write many classes without any security annotation, but still be able to use them in secure contexts with our promotion rules. We also evaluated with a benchmark [Hamann et al. 2018] precision and recall of typing sample programs with SIFO in comparison to other information flow analysis tools. Our comparison showed a similar precision as related information flow analysis tool. Only JOANA [Graf et al. 2013] and Co-Inflow [Xiang and Chong 2021] with additional annotations in the source code were more precise. In the benchmark, many samples are contrived to have an information flow that is difficult to analyze. SIFO is too strict for some of these samples and rejects them, but we argue that developers would not write real code this way with our type system. Moreover, these samples can be easily rewritten to be accepted by SIFO.

## Information Flow Security by Construction

Our overall goal is to develop programs that are functionally correct and secure. We intend to take advantage of CbC to incrementally construct programs that are guaranteed to ensure functional

correctness and information flow security. In the previous part, we introduced the SIFO type system that guarantees for information flow security in checked programs. On that basis, we establish refinement rules and propose IFBCOO,<sup>6</sup> a program development approach for information flow security. IFBCOO in combination with the classic CbC refinement rules then enables a uniform correct-by-construction approach for both functional and security properties.

With IFBCOO programs are constructed stepwise to comply with a given information flow policy (i.e., a lattice of security levels given by the developer). We start a method implementation with a provided functional and security specification in form of an IFBCOO tuple  $\{P; Q; \Gamma; T; eA\}$ . The functional specification is written as precondition  $P$  and postcondition  $Q$ . The IFBCOO tuple also has a typing context  $\Gamma ::= x_1 : T_1 \dots x_n : T_n$  that is extracted from the methods arguments with return type  $T$ . The starting abstract expression  $eA$  is typed by the composed type  $[P; Q; \Gamma; T]$ . With this type, we have all information to refine the abstract expression. We have to satisfy the functional pre- and postcondition specification  $P$  and  $Q$  which is guaranteed through the classic CbC refinement rules. We also have to find a refined expression that has the type  $T$ , and uses only variables from the context  $\Gamma$ . That the refined expression is secure is guaranteed with the new refinement rules for information flow security.

As an example, we present the field assignment rule in [Definition 3.4](#). In this rule, we omit the functional specification. An abstract expression  $eA$ , can be refined to a field assignment  $eA_0.f := eA_1$ , if the security side conditions hold. These side conditions state that the expression  $eA_0$  must be mut to allow a manipulation of the object. The assigned expression  $eA_1$  must have the same security level as the combined security level (least upper bound of the security levels) of expression  $eA_0$  and field  $f$ . The field  $f$  must be declared in the class  $C_0$  with the type  $s \text{ mdf } C$ .

#### Definition 3.4: Field Assignment

An abstract expression  $eA$  is refinable to a field assignment  $eA_0.f := eA_1$  if  $eA : [\Gamma; T]$  and  $eA_0 : [\Gamma; s_0 \text{ mut } C_0]$  and  $eA_1 : [\Gamma; s_1 \text{ mdf } C]$  and  $s \text{ mdf } C f \in \text{fields}(C_0)$  and  $s_1 = \text{lub}(s_0, s)$ .

**Example 3.5.** To give an example, we refine the abstract expression  $eA$  that is typed:

$$[\text{c.blc.blc} = 0 \ \& \ \text{y} = 1; \text{c.blc.blc} = 1; \Gamma; \text{high imm int}]$$

with  $\Gamma := \text{c} : \text{low mut Card}, \text{y} : \text{low imm int}$ . We use the class `Card` of [Listing 3.5](#). We can refine the abstract expression to a field assignment: `c.blc.blc = y`; The classic CbC assignment rule ensures that the postcondition  $Q$  is satisfied (i.e., the field has the correct value of 1). The IFBCOO refinement rule guarantees that the variable  $y$  can be assigned to the field without violating the information flow policy. The low imm variable  $y$  is promoted to high, and then assigned to the field `c.blc.blc` with the same security level. If instead the variable  $y$  is typed as mut, the promotion to a high security level is insecure, and IFBCOO prohibits the assignment.

In our work [[Runge et al. 2022a](#)], we formalized IFBCOO by introducing 13 refinement rules and proved soundness by showing that constructed programs with IFBCOO are well-typed in SIFO. We also implemented IFBCOO in CoRC. To provide the necessary information, we have to annotate fields and methods in classes, and local variables in methods with security levels and type

<sup>6</sup>IFBCOO is an acronym for Information Flow Control-by-Construction for an Object-Oriented language

modifiers. We are using the annotation types of Java to include this information without further issues. By implementing the refinement rules for security in CORC, we support CbC for functional correctness and information flow security. The feasibility of CORC is evaluated with the same case studies that are used to evaluate SIFO. The case studies were successfully implemented in CORC. During the construction of the case studies, we noticed that the constructive approach has the advantage of providing feedback at each refinement step when security policies are violated. A violation is directly observed and can be resolved. With post-hoc analysis tools, feedback is only available after writing complete methods.

## Related Work

We discuss related work to enforce secure information flow in programs. Besides security type systems, static and dynamic analysis are used. We also discuss Hoare-style program logic for information flow reasoning.

**Taint Analysis.** Static [Arzt et al. 2014; Huang et al. 2014; Graf et al. 2013; Mohr et al. 2015] and dynamic [Enck et al. 2014; Hedin et al. 2014; Roy et al. 2009; Austin and Flanagan 2009] taint analysis detects insecure information flows from tainted sources to secure sinks. These approaches analyze the direct flow of information through the assignments of values to variables and fields. JSFlow [Hedin et al. 2014], JOANA [Graf et al. 2013], and JoDroid [Mohr et al. 2015] also cover implicit information flows through conditional statements, loop statements, or dynamic dispatch. In our work with SIFO and IFBCOO, we detect both direct and implicit information flows, but we additionally proved soundness of SIFO and IFBCOO. The related taint analyses do not provide a soundness property.

Coarse-grained dynamic information flow approaches [Xiang and Chong 2021; Nadkarni et al. 2016; Jia et al. 2013] track information at the granularity of lexically or dynamically scoped sections of code. These approaches label an entire section of code with only one label, in comparison to fine-grained approaches that label every value individually. Thus, coarse-grained approaches can reduce the writing effort for developers because they do not have to annotate the complete program. For dynamic information flow control approaches, Vassena et al. [Vassena et al. 2019] showed that fine-grained and coarse-grained approaches are similar in terms of precision. For example, by using techniques such as the opaque labeled values introduced by Xiang and Chong [2021], the information flow analysis results of coarse-grained approaches are accurate. SIFO and IFBCOO are fine-grained approaches with also reduced writing effort through well-chosen standards and promotion rules. The promotion rules allow us to use classes without security annotations (i.e., the default security level is assumed) in contexts where higher security levels are required.

**Type Systems.** Type systems to control the information flow are widely studied [Sabelfeld and Myers 2003; Simonet 2003; Ferraiuolo et al. 2017; Zhang et al. 2015]. Typically, type systems are classified into flow-sensitive [Hunt and Sands 2006; Li and Zhang 2017] or flow-insensitive [Myers 1999; Volpano et al. 1996] systems. A flow-insensitive type system gets the same result for a program independent of the sequence of statements (e.g.,  $low = high$ ;  $low = \emptyset$ ; and  $low = \emptyset$ ;  $low = high$ ; are both discarded because of the insecure assignment of  $low = high$ ). A flow-sensitive type system



would type the first version as secure, as the confidential data is overwritten by the second statement. SIFO is flow insensitive, but as shown by Hunt and Sands [2006], any program that is typable by a flow-sensitive type system can be transformed to be typable by a flow-insensitive type system.

With SIFO, we focused on secure type systems for object-oriented languages [Sabelfeld and Myers 2003; Banerjee and Naumann 2002; Sun et al. 2004; Myers 1999; Strecker 2003; Barthe and Serpette 1999; Barthe et al. 2007]. The most important work to compare with is Jif [Myers 1999]. Jif treats aliases of objects differently to SIFO. Jif does not use any kinds of regions or alias analysis to reason about bounded side effects. Therefore, Jif pessimistically discards programs introducing aliases because possible leaks through the usage of aliased objects cannot be estimated. With SIFO, we only restrict the introduction of insecure aliases. When we reason about immutable or encapsulated objects, we can be sure that no information leak is possible. A more detailed comparison to Jif is available in [Runge et al. 2022c]. There, we discuss how the additional features of Jif, that are not available in the core of SIFO, can be emulated in our implementation.

**Logics for Information Flow.** Hoare-style program logics are used to reason about secure information flow. To analyze sequential and parallel programs, Andrews and Reitman [1980] encode information flow in Hoare-style formulas and proof validity of these formulas. Amtoft et al. [2006] and Amtoft and Banerjee [2004] also use Hoare-style program logic in combination with abstract interpretation to analyze the information flow. Their work is the foundation for specifying and analyzing information flow in SPARK Ada [Amtoft et al. 2008]. Barthe et al. [2004] and Darvas et al. [2005] analyze the information flow of programs by formalizing information flow properties in a programming logic using self-composition of programs and checking the formalized properties with standard program verification tools. Similarly, Beckert et al. [2013] analyze the information flow of Java programs using self-composition and the program verification tool KEY. Küsters et al. [2015] present a hybrid approach by combining JOANA [Graf et al. 2013] and verification with KEY [Ahrendt et al. 2016] to analyze the information flow. By combining self-composition of programs with a type system, Terauchi and Aiken [2005] proposed an approach that benefits from both. With SIFO, we are relying on a type system to ensure correct information flow, and with IFBCOO, we established refinement rules based on the type system. SIFO and IFBCOO are both used to construct secure programs. Hoare-style program logics for secure information flow are post-hoc analysis techniques.

## Conclusion

The goal of this contribution [Runge et al. 2020; Runge et al. 2022c; Runge et al. 2022a] is to establish a CbC approach for functionally correct and secure programs. We achieved this in two steps. First, we developed SIFO that supports information flow control for an object-oriented language. A type system checks that the program satisfies a provided information flow policy. We proved soundness of SIFO, implemented SIFO, and showed in a feasibility study the advantages of using immutability analysis for the detection of information flow leaks. We also measured similar precision and recall on the sample programs with SIFO in comparison to information flow analysis tools. Second, we established refinement rules on the basis of SIFO to incrementally construct secure programs. We also proved soundness of IFBCOO by showing that constructed programs are typable in SIFO. IFBCOO is implemented in CoRC and also evaluated with a feasibility study where we se-

curely implemented four case studies. With this contribution we addressed **Challenge 3**, and we can positively answer *Research Question RQ<sub>4</sub> – Correctness-by-Construction for Information Flow Security* that it is possible to establish a uniform correct-by-construction program development approach. IFBCOO refinement rules in combination with classic CbC refinement rules guaranteed information flow security and functional correctness in each refinement step.





# 4. Conclusion

We conclude this thesis with a summary and a discussion of our contributions. We also present the most interesting directions for future work.

## 4.1. Discussion of Research Questions

In this thesis, we provided the concept and the implementation of the CoRC tool support, which enables correct-by-construction program development for functionally correct and secure programs. In the following paragraphs, we summarize our contributions grouped by our four research questions, and from which we derive an answer to our main research question.

### Research Question RQ<sub>1</sub> – Tool Support for Functional Correctness-by-Construction

Our first research question *How can we support a correct-by-construction program development approach for functionally correct software?* is addressed with our first contribution [Runge et al. 2019a] (see Section A.1). We focused on supporting correctness-by-construction as proposed by Kourie and Watson [2012]. We investigated several other CbC approaches [Dijkstra 1976; Gries 1987; Morgan 1994; Back 2009], but concluded that CbC as proposed by Kourie and Watson [2012] is suited best for our purpose because of their focus on comprehensibility and applicability, which are important properties for developers to be willing to use this approach. Since there was no tool support for this CbC approach, we stated requirements to support CbC and developed the open source tool CoRC according to these requirements. CoRC consists of a graphical and a textual editor to develop programs using CbC. The correctness of developed programs is ensured by translating the side conditions of applied refinement rules to proof obligations for the program verifier KEY. KEY then discharges these proof obligations automatically.

With our implementation of CoRC, we answer the first research question. In the evaluation, we discussed that CoRC meets the stated requirements to support correctness-by-construction. We also demonstrated that sufficiently complex case studies can be successfully constructed using CoRC. In fact, we discovered six samples in the case studies that could only be verified through CoRC's fine-step construction approach and not automatically with traditional post-hoc verification. This highlights a major benefit of using correctness-by-construction implemented in CoRC. Therefore, we can conclude that CoRC is a suitable tool for correct-by-construction program development.

### Research Question RQ<sub>2</sub> – Usability of Tool-Supported Correctness-by-Construction

Our second research question *How usable is the correct-by-construction program development approach with CoRC?* is addressed with our second contribution [Runge et al. 2019b; Runge et al.

2021] (see Section A.2 and Section A.3). We conducted two user studies to evaluate CORC’s usability, which is a key criterion for CORC to become an established development tool. In these user studies, we assessed the differences in program development using (1) CORC and (2) state-of-the-art post-hoc verification (PhV) with KEY.

In both user studies, we observed that the participants make more errors with CbC than with PhV, but we have a similar number of correctly specified and proven programs with both approaches. In terms of actual defects, we found similar types with both approaches. For example, an inappropriate invariant or a slightly incorrect guard prevented the verification of a loop statement. Through the analysis of the intermediate development results, we discovered a trial-and-error approach to construct and verify programs with PhV when the participant could not immediately detect the reason why a program was not provable. Surprisingly, the participants also changed correct code when they could not verify the program. With CbC, the participants started with a structured refinement process (e.g., they specified the program first, or they specified and refined the program simultaneously), but they abandoned the structured process as soon as they got stuck. Even though the participants made more errors with CORC, they answered the user experience questionnaire in favor of CORC with a significant difference for the measurements stimulation and novelty of the tool. In the open questions of the interview, the participants highlighted that they like the structured reasoning of CbC, and the fine-grained feedback through the separate proofs of each applied refinement rule. Without feedback, locating of defects was frustrating or impossible for the participants. They also stated that the structured CbC process helps to think about the specification and corner cases of the program before writing the code. In favor of PhV, mostly the familiar environment was mentioned.

With the results of the user studies, we answer the second research question. The participants rated that CORC is more suitable for finding defects than PhV with KEY. They also emphasized the better feedback of CORC due to the structured construction with refinement rules and their direct proof of correctness. Therefore, we conclude that CORC is usable to develop programs according to CbC. Based on the feedback from participants, we also expect that users will make fewer errors as they become more familiar with CORC.

### Research Question RQ<sub>3</sub> – Alternatives to Rule-based Correctness-by-Construction

Our third research question *What alternative correct-by-construction program development approaches exist, and how do they compare to rule-based correctness-by-construction?* is addressed with our third contribution [Runge et al. 2022b; Runge et al. 2023] (see Section A.4 and Section A.5). The classic CbC approach by Kourie and Watson [2012] has a limited flexibility in program development. In this contribution, we proposed two alternative CbC approaches, namely CBC-BLOCK and TRAITCBC, that address this rigid program development. CBC-BLOCK adds two new refinement rules which allow more flexible program development. Instead of just adding one statement, a block (i.e., a sequence of statements) can be added to the program. CBC-BLOCK is implemented as extension of CORC and evaluated with a user study. In the user study, the participants appreciated the increased flexibility with the block refinement rules and stated that they would use this extension for program development. TRAITCBC is an alternative CbC approach that uses composition of methods in traits to offer an incremental development approach. The ad-

vantage of `TRAITCbC` is that it does not need refinement rules to ensure correctness. `TRAITCbC` is evaluated to be feasible by constructing several case studies.

By discussing the characteristics of `CbC-BLOCK` and `TRAITCbC` in comparison to classic `CbC`, we answer the third research question. All approaches are tool supported and guarantee the correctness of a program under development in each construction step. Both, `CbC-BLOCK` and `TRAITCbC`, allow more flexible program construction. `CbC-BLOCK` adds new refinement rules for this purpose, but `TRAITCbC` does not require any refinement rules and relies only on correct method composition. Therefore, we proposed alternative `CbC` program development approaches with the same advantages as classic `CbC`, but without the disadvantage of rigid program development.

## Research Question RQ4 – Correctness-by-Construction for Information Flow Security

Our fourth research question *How can we support the development of programs with correctness-by-construction that are functionally correct and satisfy information flow security?* is addressed with our fourth contribution [Runge et al. 2022c; Runge et al. 2020; Runge et al. 2022a] (see Section A.6, Section A.7, and Section A.8). In this contribution, we developed `CbC` for information flow security to establish our goal of a uniform correct-by-construction program development approach for functional correctness and information flow security.

As part of our contribution, we developed `SIFO`, a new type system for information flow security. We established typing and reduction rules, and we proved soundness of `SIFO`. A program typable in `SIFO` ensures noninterference. We noticed that large parts of a program can be written without any security annotation, but still be used in secure context through well-chosen standards and security promotion rules. We developed `CbC`-style refinement rules based on `SIFO` and established `IFBCOO` for secure program construction. In total, we proposed 13 refinement rules that allow to construct object-oriented programs enriched with type modifiers. We proved soundness of the development approach by showing that a constructed program is typable in `SIFO`. `IFBCOO` is implemented in `CORC`.

With the proposed `IFBCOO` approach, we answer the fourth research question. By having classic `CbC` refinement rules and refinement rules for information flow security integrated in `CORC`, we provided a uniform development approach to guarantee both properties; functional correctness and information flow security. We also showed that it is feasible to develop secure programs by construction by implementing case studies in `CORC`. Therefore, we proposed a new applicable `CbC` approach for a non-functional property. For future work, we investigate how other non-functional properties can be ensured with `CbC`.

## 4.2. Discussion of the Main Research Question

The goal of this thesis is to address our main research question: *How can we enable and support a correct-by-construction program development approach for functionally correct and secure software?* In our contributions, we introduced `CORC`, tool support for correctness-by-construction as proposed by Kourie and Watson [2012]. This tool enables the construction of functionally correct programs

through the correctness guarantees of the provided refinements rules. We enriched the flexibility of program development with new refinement rules to introduce blocks of statements instead of just single statements. Security is ensured by the proposed IFBCOO refinement rules, which prevent explicit or implicit information leaks in the program. CORC is evaluated by implementing several cases where we measured a similar verification effort as with post-hoc verification, but we were able to verify more samples with CbC than with post-hoc verification. We evaluated the usability of CORC with two user studies. Participants stated a preference for CbC over post-hoc verification to detect and correct defects. They also emphasized that they like to construct programs using the structured reasoning of CbC, rather than hacking programs into correctness with post-hoc verification. However, higher defect rates with CORC indicate that the participants need more experience with the new tool.

In summary, we established and supported a correct-by-construction program development approach for functionally correct and secure programs. Furthermore, we evaluated its feasibility and confirmed the usability of the tool by conducting two user studies. Consequently, we can positively answer our main research question with the implementation and evaluation of CORC. However, further evaluation is needed to determine whether CbC is an alternative to state-of-the-art program construction with post-hoc verification and post-hoc information flow analysis. In our user studies, we found qualitative usability results in favor of CbC, but these results need to be confirmed in larger and longer (industrial) studies. If people continue to use post-hoc verification and post-hoc analysis, we encourage to use it in concert with CbC to take advantage of both approaches. Depending on the nature of the problem and prior knowledge of the developers, software can be developed and verified with one or a combination of the approaches. If necessary (e.g., specified by a standard [RTCA DO-178C 2011]), a program constructed in CORC can be easily verified post-hoc, since the program is fully specified and should be amenable to automatic post-hoc verification [Watson et al. 2016], especially since CORC is based on a PhV tool.

### 4.3. Ongoing and Future Work

The presented correct-by-construction program development approach offers many opportunities for future work. We present the most interesting directions in the following.

#### Extensions of CORC

With CORC, we established the foundation for tool supported correctness-by-construction. This implementation inspired us to expand the scope of correct-by-construction program development into additional domains. With VARCORC [Bordis et al. 2020a; Bordis et al. 2022b], the correct construction of software product lines is supported. Instead of developing monolithic systems, VARCORC ensures the correctness of feature-oriented software [Pohl et al. 2005; Batory 2004]. With ARCHICORC [Knüppel et al. 2020], we scale CbC to the development of correct component-based architectures [Sametinger 1997; Szyperski et al. 2002]. ARCHICORC supports the creation of a correct repository of components, where the implementations are accessed through explicit interfaces for standardized integration into personal projects. WEBCORC [Runge et al. 2021] is implemented as web-frontend for CORC to increase the reach and influence of our work on CbC. This tool is also

used to conduct user studies remotely. The parallel efforts to extend CORC are an integral part of the ongoing work to scale CbC to software product lines with VARCORC and component-based architectures with ARCHICORC, and to provide an easy access and demonstration of CbC with WEBCORC.

## Case Studies for the Evaluation of CbC

So far, we evaluated CORC on small to mid-sized case studies to focus on feasibility of our concepts and the usability of the tool itself. To assess the applicability of CORC for realistically-sized software systems, larger-scale case studies are needed. This means, the scalability of the approach for larger methods, and also the ability whether CbC can be integrated into software engineering processes with development teams could be evaluated with these large-scale case studies. Moreover, the specification, programming, and verification effort could be more accurately evaluated with more case studies. For example, it could be measured how long developers need to specify and develop correct programs with CbC. An interesting investigation would be whether a positive effect compared to standard development with post-hoc verification could be observed. We already measured the verification effort in our work, but more case studies could reveal significant differences between the approaches.

## User Studies with Experts

We could gain new insights into the benefits and drawbacks of CbC and CORC by conducting further user studies with experts in post-hoc verification and CbC. When experts use the tools, we assume that they avoid simple errors that we noticed in our previous studies (e.g., a missing initialization of a variable). The user studies could include larger algorithms over longer periods of time, so that not only simple tasks are solved with CORC. With larger tasks and experts, we might gain further insight into the usability of CORC. With these results, we could determine whether CbC as implemented in CORC is a viable approach to develop correct and secure programs, or whether CORC needs to be improved. With CORC, we created the basis to evaluate CbC, but there are not many CORC experts for user studies yet. Developers must be trained by lectures or other events, such as tutorials [Schaefer et al. 2021].

## Tool Improvements

The CORC tool supports the construction of correct Java programs, but we identified potential to improve the usability of CORC in several ways. Currently, a developer has to provide the specification and apply each refinement rule manually. The integration of program or specification synthesis tools [Gulwani et al. 2017; Chen et al. 2015] can support the developer to find correct programs and specifications. For example, simple leaf nodes (i.e., assignment statements) could be generated automatically. A loop invariant could also be generated. These generated parts should only be treated as suggestions for developers since CbC is about correct program construction and not about correct program generation.

CORC could be extended to support other languages, such as C# or Dafny. As a result, it would be possible to develop programs correct by construction in other languages as well. To achieve that, the editor has to support the syntax of the programming language and its corresponding

specification language. To check for functional correctness, the side conditions of the refinement rule have to be translated to proof obligations checkable by a program verifier for the new language. The current format of the generated proof obligations is tailored to KEY and should be adopted. For information flow security, the new language has to support the language features and type modifiers of SIFO. For our implementation in CORC, we annotate Java methods, fields, and variables with security labels and type modifiers.

## X-by-Construction

With IFBCOO, we demonstrated that secure programs can be constructed with a correct-by-construction program development approach. However, we did not consider further non-functional properties, such as timing behavior or resource consumption yet. For example, CbC would be a promising approach for real-time applications if a program constructed with CbC is guaranteed to be faster than a (worst-case) execution time. With the term X-by-construction [Beek et al. 2018] (e.g., timing-by-construction), we summarize CbC approaches for any non-functional property. To establish an X-by-construction approach, it must be explored what is necessary to develop refinement rules for a specific property. The following questions should be answered to establish a guideline for the creation of X-by-construction approaches. Is it always feasible to transform typing rules into CbC-style refinement rules or only for a certain kind of type system? Can any post-hoc analysis technique be translated to a constructive refinement system? Do we sometimes have to develop refinement rules from scratch? In which form must information about non-functional properties be available (e.g., Hoare-style logic)? If a guideline for implementing X-by-construction is created based on these questions, developers can take advantage of CbC to ensure various non-functional properties.



# Bibliography

- Abrial, J.-R. (2010). *Modeling in Event-B - System and Software Engineering*. Cambridge University Press.
- Abrial, J.-R., M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin (2010). “Rodin: An Open Toolset for Modelling and Reasoning in Event-B”. *International Journal on Software Tools for Technology Transfer* 12.6, pp. 447–466.
- Agda (n.d.). *Agda Development Team. The Agda wiki, 2007-2021*. <http://wiki.portal.chalmers.se/agda/pmwiki.php>. Accessed: 2021-06-18.
- Ahrendt, W., B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich (2016). *Deductive Software Verification – The KeY Book: From Theory to Practice*. Vol. 10001. Springer.
- Amighi, A., S. Blom, S. Darabi, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski (2014). “Verification of Concurrent Systems with VerCors”. In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Vol. 8483. Lecture Notes in Computer Science. Springer, pp. 172–216.
- Amtoft, T., S. Bandhakavi, and A. Banerjee (2006). “A Logic for Information Flow in Object-Oriented Programs”. In: *Proceedings of the ACM on Programming Languages*, pp. 91–102.
- Amtoft, T. and A. Banerjee (2004). “Information Flow Analysis in Logical Form”. In: *International Static Analysis Symposium*. Vol. 3148. Lecture Notes in Computer Science. Springer, pp. 100–115.
- Amtoft, T., J. Hatcliff, E. Rodriguez, Robby, J. Hoag, and D. A. Greve (2008). “Specification and Checking of Software Contracts for Conditional Information Flow”. In: *International Symposium on Formal Methods*. Springer, pp. 229–245.
- Andrews, G. R. and R. P. Reitman (1980). “An Axiomatic Approach to Information Flow in Programs”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1, pp. 56–76.
- Arzt, S., S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Outeau, and P. D. McDaniel (2014). “FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Vol. 49. 6. ACM, pp. 259–269.
- Austin, T. H. and C. Flanagan (2009). “Efficient Purely-Dynamic Information Flow Analysis”. In: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. ACM, pp. 113–124.
- Back, R.-J. (2009). “Invariant Based Programming: Basic Approach and Teaching Experiences”. *Formal Aspects of Computing* 21.3, pp. 227–244.

- Back, R.-J., J. Eriksson, and M. Myreen (2007). “Testing and Verifying Invariant Based Programs in the SOCOS Environment”. In: *International Conference on Tests and Proofs (TAP)*. Vol. 4454. Lecture Notes in Computer Science. Springer, pp. 61–78.
- Back, R.-J. and J. Wright (2012). *Refinement Calculus: A Systematic Introduction*. Springer Science & Business Media.
- Banerjee, A. and D. A. Naumann (2002). “Secure Information Flow and Pointer Confinement in a Java-like Language”. In: *Proceedings of the 15th IEEE workshop on Computer Security Foundations*. Vol. 2, p. 253.
- Barnes, J. G. P. (2003). *High Integrity Software: The Spark Approach to Safety and Security*. Pearson Education.
- Barnett, M., B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino (2005). “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *International Symposium on Formal Methods for Components and Objects*. Vol. 4111. Lecture Notes in Computer Science. Springer, pp. 364–387.
- Barnett, M., M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter (June 2011). “Specification and Verification: The Spec# Experience”. *Communication of the ACM* 54.6, pp. 81–91.
- Barnett, M., K. R. M. Leino, and W. Schulte (2004). “The Spec# Programming System: An Overview”. In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer, pp. 49–69.
- Barthe, G., P. R. D’Argenio, and T. Rezk (2004). “Secure Information Flow by Self-Composition”. In: *Computer Security Foundations Symposium (CSF)*. IEEE, pp. 100–114.
- Barthe, G., D. Pichardie, and T. Rezk (2007). “A Certified Lightweight Non-Interference Java Bytecode Verifier”. In: *European Symposium on Programming*. Vol. 4421. Lecture Notes in Computer Science. Springer, pp. 125–140.
- Barthe, G. and B. P. Serpette (1999). “Partial Evaluation and Non-Interference for Object Calculi”. In: *International Symposium on Functional and Logic Programming*. Lecture Notes in Computer Science. Springer, pp. 53–67.
- Batory, D. (2004). “Feature-Oriented Programming and the AHEAD Tool Suite”. In: *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, pp. 702–703.
- Beckert, B., D. Bruns, V. Klebanov, C. Scheben, P. H. Schmitt, and M. Ulbrich (2013). “Information Flow in Object-Oriented Software”. In: *International Symposium on Logic-Based Program Synthesis and Transformation*. Lecture Notes in Computer Science. Springer, pp. 19–37.
- Beckert, B., S. Grebing, and F. Böhl (2014a). “A Usability Evaluation of Interactive Theorem Provers Using Focus Groups”. In: *International Conference on Software Engineering and Formal Methods*. Vol. 8938. Lecture Notes in Computer Science. Springer, pp. 3–19.



- Beckert, B., S. Grebing, and F. Böhl (2014b). “How to Put Usability into Focus: Using Focus Groups to Evaluate the Usability of Interactive Theorem Provers”. *Electronic Proceedings in Theoretical Computer Science* 167, pp. 4–13.
- Beek, M. H. ter, L. Cleophas, I. Schaefer, and B. W. Watson (2018). “X-by-Construction”. In: *International Symposium on Leveraging Applications of Formal Methods*. Cham: Springer International Publishing, pp. 359–364.
- Bell, D. E. and L. J. La Padula (1976). *Secure Computer System: Unified Exposition and Multics Interpretation*. Tech. rep. MITRE Corp Bedford MA.
- Bettini, L., F. Damiani, and I. Schaefer (2010). “Implementing Software Product Lines Using Traits”. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 2096–2102.
- Bettini, L., F. Damiani, I. Schaefer, and F. Strocchio (2013). “TRAITRECORDJ: A Programming Language with Traits and Records”. *Science of Computer Programming* 78.5, pp. 521–541.
- Biba, K. J. (1977). *Integrity Considerations for Secure Computer Systems*. Tech. rep. MITRE Corp Bedford MA.
- Bono, V., E. Mensa, and M. Naddeo (2014). “Trait-Oriented Programming in Java 8”. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pp. 181–186.
- Bordis, T., L. Cleophas, A. Kittelmann, T. Runge, I. Schaefer, and B. W. Watson (2022a). “Re-CorC-ing KeY: Correct-by-Construction Software Development Based on KeY”. In: *The Logic of Software. A Tasting Menu of Formal Methods*. Vol. 13360. Lecture Notes in Computer Science. Springer. DOI: [10.1007/978-3-031-08166-8\\_5](https://doi.org/10.1007/978-3-031-08166-8_5).
- Bordis, T., T. Runge, A. Knüppel, T. Thüm, and I. Schaefer (2020a). “Variational Correctness-by-Construction”. In: *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS)*. ACM, 7:1–7:9. DOI: [10.1145/3377024.3377038](https://doi.org/10.1145/3377024.3377038).
- Bordis, T., T. Runge, and I. Schaefer (2020b). “Correctness-by-Construction for Feature-Oriented Software Product Lines”. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. ACM, pp. 22–34. DOI: [10.1145/3425898.3426959](https://doi.org/10.1145/3425898.3426959).
- Bordis, T., T. Runge, D. Schultz, and I. Schaefer (2022b). “Family-Based and Product-Based Development of Correct-by-Construction Software Product Lines”. *Journal of Computer Languages*, p. 101119. DOI: [10.1016/j.col.2022.101119](https://doi.org/10.1016/j.col.2022.101119).
- Cataño, N. (2019). “Teaching Formal Methods: Lessons Learnt from Using Event-B”. In: *Formal Methods Teaching*. Vol. 11758. Lecture Notes in Computer Science. Springer, pp. 212–227.
- Chang, C.-L. and R. C.-T. Lee (2014). *Symbolic Logic and Mechanical Theorem Proving*. Academic press.
- Chen, H.-Y., C. David, D. Kroening, P. Schrammel, and B. Wachter (2015). “Synthesising Interprocedural Bit-Precise Termination Proofs (T)”. In: *International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 53–64.

- Clarke, E. M., T. A. Henzinger, H. Veith, and R. Bloem (2018). *Handbook of Model Checking*. Vol. 10. Springer.
- Clements, P. and L. Northrop (2002). *Software Product Lines: Practices and Patterns*. Addison-Wesley.
- Cohen, E., M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies (2009). “VCC: A Practical System for Verifying Concurrent C”. In: *International Conference on Theorem Proving in Higher Order Logics*. Vol. 5674. Lecture Notes in Computer Science. Springer, pp. 23–42.
- Cok, D. R. (2011). “OpenJML: JML for Java 7 by Extending OpenJDK”. In: *NASA Formal Methods Symposium*. Vol. 6617. Lecture Notes in Computer Science. Springer, pp. 472–479.
- Coq (n.d.). *Coq Development Team. The Coq Proof Assistant, 1989-2021*. <http://coq.inria.fr>. Accessed: 2021-06-18.
- Creuse, L., C. Dross, C. Garion, J. Hugues, and J. Huguet (2019). “Teaching Deductive Verification Through Frama-C and SPARK for Non Computer Scientists”. In: *Formal Methods Teaching*. Vol. 11758. Lecture Notes in Computer Science. Springer, pp. 23–36.
- Cuoq, P., F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski (2012). “Frama-C”. In: *International Conference on Software Engineering and Formal Methods*. Vol. 7504. Lecture Notes in Computer Science. Springer, pp. 233–247.
- Damiani, F., J. Dovland, E. B. Johnsen, and I. Schaefer (2014). “Verifying Traits: An Incremental Proof System for Fine-Grained Reuse”. *Formal Aspects of Computing* 26.4, pp. 761–793.
- Darvas, Á., R. Hähnle, and D. Sands (2005). “A Theorem Proving Approach to Analysis of Secure Information Flow”. In: *International Conference on Security in Pervasive Computing*. Vol. Lecture Notes in Computer Science. 3450. Springer, pp. 193–209.
- Denning, D. E. (1976). “A Lattice Model of Secure Information Flow”. *Communication of the ACM* 19.5, pp. 236–243.
- Dijkstra, E. W. (1975). “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. *Communications of the ACM* 18.8, pp. 453–457.
- Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice Hall.
- Dijkstra, E. W. (1972). “Notes on Structured Programming”. In: *Structured Programming*. Academic Press Inc., pp. 1–82.
- Divasón, J. and A. Romero (2019). “Using Krakatoa for Teaching Formal Verification of Java Programs”. In: *Formal Methods Teaching*. Vol. 11758. Lecture Notes in Computer Science. Springer, pp. 37–51.
- Ducasse, S., O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black (2006). “Traits: A Mechanism for Fine-Grained Reuse”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.2, pp. 331–388.

- Enck, W., P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth (June 2014). “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”. *ACM Transactions on Computer Systems (TOCS)* 32.2.
- Ettinger, R. (2021). “Lessons of Formal Program Design in Dafny”. In: *Formal Methods Teaching*. Vol. 13122. Lecture Notes in Computer Science. Springer, pp. 84–100.
- Ferraiuolo, A., W. Hua, A. C. Myers, and G. E. Suh (2017). “Secure Information Flow Verification with Mutable Dependent Types”. In: *Design Automation Conference (DAC)*. IEEE, pp. 1–6.
- Flatt, M., S. Krishnamurthi, and M. Felleisen (1998). “Classes and Mixins”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 171–183.
- Gentzen, G. (1935). “Untersuchungen über das logische Schliessen. II”. *Mathematische Zeitschrift* 39.1, pp. 405–431.
- Gerhart, S. L. (1975). “Correctness-Preserving Program Transformations”. In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*, pp. 54–66.
- Giannini, P., M. Servetto, E. Zucca, and J. Cone (2019). “Flexible Recovery of Uniqueness and Immutability”. *Theoretical Computer Science* 764, pp. 145–172.
- Gouw, S. d., J. Rot, F. S. d. Boer, R. Bubel, and R. Hähnle (2015). “OpenJDKs Java.utils.Collection.sort() is Broken: The Good, The Bad and The Worst Case”. In: *International Conference on Computer Aided Verification*. Vol. 9206. Lecture Notes in Computer Science. Springer, pp. 273–289.
- Graf, J., M. Hecker, and M. Mohr (2013). “Using JOANA for Information Flow Control in Java Programs - A Practical Guide”. In: *Proceedings of the 6th Working Conference on Programming Languages (ATPS’13)*. Lecture Notes in Informatics (LNI) 215. Springer, pp. 123–138.
- Gries, D. (1987). *The Science of Programming*. Springer.
- Gulwani, S., S. Jha, A. Tiwari, and R. Venkatesan (2010). *Component Based Synthesis Applied to Bitvector Programs*. Tech. rep. Citeseer.
- Gulwani, S., O. Polozov, R. Singh, et al. (2017). “Program Synthesis”. *Foundations and Trends in Programming Languages* 4.1-2, pp. 1–119.
- Hall, A. and R. Chapman (2002). “Correctness by Construction: Developing a Commercial Secure System”. *IEEE Software* 19.1, pp. 18–25.
- Hall, R. J. (2005). “Fundamental Nonmodularity in Electronic Mail”. *Automated Software Engineering* 12.1, pp. 41–79.
- Hamann, T., M. Herda, H. Mantel, M. Mohr, D. Schneider, and M. Tasch (2018). “A Uniform Information-Flow Security Benchmark Suite for Source Code and Bytecode”. In: *Nordic Conference on Secure IT Systems*. Springer, pp. 437–453.
- Harel, D., D. Kozen, and J. Tiuryn (2000). “Dynamic Logic”. In: *Foundations of Computing*. MIT Press.

- Hedin, D., A. Birgisson, L. Bello, and A. Sabelfeld (2014). “JSFlow: Tracking Information Flow in JavaScript and Its APIs”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*.
- Heisel, M. (1992). “Formalizing and Implementing Gries’ Program Development Method in Dynamic Logic”. *Science of Computer Programming* 18.1, pp. 107–137.
- Hentschel, M. (2016). “Integrating Symbolic Execution, Debugging and Verification”. PhD thesis. Technische Universität Darmstadt.
- Hentschel, M., R. Hähnle, and R. Bubel (2016a). “An Empirical Evaluation of Two User Interfaces of an Interactive Program Verifier”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 403–413.
- Hentschel, M., R. Hähnle, and R. Bubel (2016b). “Can Formal Methods Improve the Efficiency of Code Reviews?” In: *International Conference on Integrated Formal Methods*. Vol. 9681. Lecture Notes in Computer Science. Springer, pp. 3–19.
- Hoare, C. A. R. (1969). “An axiomatic basis for computer programming”. *Communications of the ACM* 12.10, pp. 576–580. DOI: <https://doi.org/10.1145/363235.363259>.
- Hoare, C. A. R. (1971). “Procedures and Parameters: An Axiomatic Approach”. In: *Symposium on Semantics of Algorithmic Languages*. Springer, pp. 102–116.
- Huang, W., Y. Dong, and A. Milanova (2014). “Type-based Taint Analysis for Java Web Applications”. In: *International Conference on Fundamental Approaches to Software Engineering*. Vol. 8411. Lecture Notes in Computer Science. Springer, pp. 140–154.
- Hunt, S. and D. Sands (Jan. 2006). “On Flow-Sensitive Security Types”. *SIGPLAN Not.* 41.1, pp. 79–90.
- Igarashi, A., B. C. Pierce, and P. Wadler (2001). “Featherweight Java: A Minimal Core Calculus for Java and GJ”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.3, pp. 396–450.
- ISO/IEC 25010 (2011). *Software Considerations in Airborne Systems and Equipment Certification*.
- Jacobs, B., J. Smans, and F. Piessens (2010). “A Quick Tour of the VeriFast Program Verifier”. In: *Asian Symposium on Programming Languages And Systems*. Vol. 6461. Lecture Notes in Computer Science. Springer, pp. 304–311.
- Jia, L., J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake (2013). “Run-time Enforcement of Information-Flow Properties on Android”. In: *European Symposium on Research in Computer Security*. Springer, pp. 775–792.
- Johnson, B., Y. Song, E. Murphy-Hill, and R. Bowdidge (2013). “Why Don’t Software Developers Use Static Analysis Tools to Find Bugs?” In: *International Conference on Software Engineering (ICSE)*. IEEE Press, pp. 672–681.
- Khazeev, M., V. Rivera, M. Mazzara, and L. Johard (2016). “Initial Steps Towards Assessing the Usability of a Verification Tool”. In: *International Conference in Software Engineering for Defence Applications*. Vol. 717. Advances in Intelligent Systems and Computing. Springer, pp. 31–40.

- Knüppel, A., T. Runge, and I. Schaefer (2020). “Scaling Correctness-by-Construction”. In: *International Symposium on Leveraging Applications of Formal Methods*. Ed. by T. Margaria and B. Steffen. Vol. 12476. Lecture Notes in Computer Science. Springer, pp. 187–207. DOI: [10.1007/978-3-030-61362-4\\_10](https://doi.org/10.1007/978-3-030-61362-4_10).
- Knüppel, A., T. Thüm, C. Pardylla, and I. Schaefer (2018). “Experience Report on Formally Verifying Parts of OpenJDK’s API with KeY”. In: *Proceedings of the International Workshop on Formal Integrated Development Environment (F-IDE)*. Ed. by P. Masci, R. Monahan, and V. Prevosto. Vol. 284. EPTCS, pp. 53–70. DOI: [10.4204/EPTCS.284.5](https://doi.org/10.4204/EPTCS.284.5).
- Kourie, D. G. and B. W. Watson (2012). *The Correctness-by-Construction Approach to Programming*. Springer Science & Business Media.
- Kovács, L. and A. Voronkov (2013). “First-Order Theorem Proving and Vampire”. In: *Proc. of the Intl. Conference on Computer Aided Verification (CAV)*. Springer, pp. 1–35.
- Küsters, R., T. Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr (2015). “A Hybrid Approach for Proving Noninterference of Java Programs”. In: *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE, pp. 305–319.
- Laugwitz, B., T. Held, and M. Schrepp (Nov. 2008). “Construction and Evaluation of a User Experience Questionnaire”. In: *Symposium of the Austrian HCI and Usability Engineering Group*. Vol. 5298. Lecture Notes in Computer Science, pp. 63–76.
- Leavens, G. T., A. L. Baker, and C. Ruby (1998). “JML: a Java Modeling Language”. In: *Formal Underpinnings of Java Workshop (at OOPSLA98)*. Citeseer, pp. 404–420.
- Leavens, G. T., A. L. Baker, and C. Ruby (2006). “Preliminary Design of JML: A Behavioral Interface Specification Language for Java”. *ACM SIGSOFT Software Engineering Notes* 31.3, pp. 1–38. DOI: <https://doi.org/10.1145/1127878.1127884>.
- Leino, K. R. M. (1995). *Toward Reliable Modular Programs*. California Institute of Technology.
- Leino, K. R. M. (2010). “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Vol. 6355. Lecture Notes in Computer Science. Springer, pp. 348–370.
- Li, P. and D. Zhang (2017). “Towards a Flow- and Path-Sensitive Information Flow Analysis”. In: *Computer Security Foundations Symposium (CSF)*. IEEE, pp. 53–67.
- Liquori, L. and A. Spiwack (2008). “FeatherTrait: A Modest Extension of Featherweight Java”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30.2, pp. 1–32.
- Liskov, B. H. and J. Guttag (1986). *Abstraction and Specification in Program Development*. Vol. 180. MIT press Cambridge.
- Liskov, B. H. and J. M. Wing (1994). “A Behavioral Notion of Subtyping”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6, pp. 1811–1841.

- Liu, S., K. Takahashi, T. Hayashi, and T. Nakayama (2009). “Teaching Formal Methods in the Context of Software Engineering”. *ACM SIGCSE Bulletin* 41.2, pp. 17–23.
- Manna, Z. and R. Waldinger (1980). “A Deductive Approach to Program Synthesis”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1, pp. 90–121.
- Meyer, B. (1988). “Eiffel: A Language and Environment for Software Engineering”. *Journal of Systems and Software* 8.3, pp. 199–246.
- Meyer, B. (1992). “Applying “Design by Contract””. *Computer* 25.10, pp. 40–51.
- Mohr, M., J. Graf, and M. Hecker (2015). “JoDroid: Adding Android Support to a Static Information Flow Control Tool”. In: *Software Engineering (Workshops)*. Citeseer, pp. 140–145.
- Morgan, C. (1994). *Programming from Specifications*. 2nd. Prentice Hall.
- Moura, L. de, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer (2015). “The Lean Theorem Prover”. In: *Proc. of the Intl. Conference on Automated Deduction (CADE)*. Springer, pp. 378–388.
- Myers, A. C. (1999). “JFlow: Practical Mostly-Static Information Flow Control”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Antonio, Texas, USA: ACM, pp. 228–241.
- Nadkarni, A., B. Andow, W. Enck, and S. Jha (2016). “Practical DIFC Enforcement on Android”. In: *25th USENIX Security Symposium (USENIX Security 16)*, pp. 1119–1136.
- Nipkow, T., L. C. Paulson, and M. Wenzel (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer Science & Business Media.
- Oliveira, M. V. M., A. Cavalcanti, and J. Woodcock (2003). “ArcAngel: A Tactic Language for Refinement”. *Formal Aspects of Computing* 15.1, pp. 28–47.
- Oliveira, M. V. M., A. Cavalcanti, and J. Woodcock (2009). “A UTP Semantics for Circus”. *Formal Aspects of Computing* 21.1, pp. 3–32.
- Oliveira, M. V. M., A. C. Gurgel, and C. G. Castro (2008). “CRefine: Support for the Circus Refinement Calculus”. In: *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*. IEEE, pp. 281–290.
- Pearce, D. J. and L. Groves (2013). “Whiley: A Platform for Research in Software Verification”. In: *International Conference on Software Language Engineering*. Vol. 8225. Lecture Notes in Computer Science. Springer, pp. 238–248.
- Petiot, G., N. Kosmatov, B. Botella, A. Giorgetti, and J. Julliand (2016). “Your Proof Fails? Testing Helps to Find the Reason”. In: *International Conference on Tests and Proofs*. Vol. 9762. Lecture Notes in Computer Science. Springer, pp. 130–150.
- Plath, M. and M. Ryan (2001). “Feature Integration Using a Feature Construct”. *Science of Computer Programming* 41.1, pp. 53–84.



- Pohl, K., G. Böckle, and F. J. v. d. Linden (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.
- Polikarpova, N., I. Kuraj, and A. Solar-Lezama (2016). “Program Synthesis from Polymorphic Refinement Types”. *ACM SIGPLAN Notices* 51.6, pp. 522–538.
- Reppy, J. and A. Turon (2006). “A Foundation for Trait-based Metaprogramming”. In: *International Workshop on Foundations and Developments of Object-Oriented Languages*.
- Roy, I., D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel (2009). “Laminar: Practical Fine-Grained Decentralized Information Flow Control”. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 63–74.
- RTCA DO-178C (2011). *Software Considerations in Airborne Systems and Equipment Certification*.
- Runge, T., T. Bordis, A. Potanin, T. Thüm, and I. Schaefer (2023). “Flexible Correct-by-Construction Programming”. *Logical Methods in Computer Science*.
- Runge, T., T. Bordis, T. Thüm, and I. Schaefer (2021). “Teaching Correctness-by-Construction and Post-hoc Verification—The Online Experience”. In: *Formal Methods Teaching Workshop*. Vol. 13122. Lecture Notes in Computer Science. Springer, pp. 101–116. DOI: [10.1007/978-3-030-91550-6\\_8](https://doi.org/10.1007/978-3-030-91550-6_8).
- Runge, T., A. Kittelmann, M. Servetto, A. Potanin, and I. Schaefer (2022a). “Information Flow Control-by-Construction for an Object-Oriented Language”. In: *International Conference on Software Engineering and Formal Methods*. Vol. 13550. Lecture Notes in Computer Science. Springer, pp. 209–226. DOI: [10.1007/978-3-031-17108-6\\_13](https://doi.org/10.1007/978-3-031-17108-6_13).
- Runge, T., A. Knüppel, T. Thüm, and I. Schaefer (2020). “Lattice-Based Information Flow Control-by-Construction for Security-by-Design”. In: *FormalISE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering*. ACM, pp. 44–54. DOI: [10.1145/3372020.3391565](https://doi.org/10.1145/3372020.3391565).
- Runge, T., A. Potanin, T. Thüm, and I. Schaefer (2022b). “Traits: Correctness-by-Construction for Free”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Vol. 13273. Lecture Notes in Computer Science. Springer, pp. 131–150. DOI: [10.1007/978-3-031-08679-3\\_9](https://doi.org/10.1007/978-3-031-08679-3_9).
- Runge, T., I. Schaefer, L. Cleophas, T. Thüm, D. Kourie, and B. W. Watson (2019a). “Tool Support for Correctness-by-Construction”. In: *International Conference on Fundamental Approaches to Software Engineering*. Vol. 11424. Lecture Notes in Computer Science. Springer, pp. 25–42. DOI: [10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2).
- Runge, T., M. Servetto, A. Potanin, and I. Schaefer (2022c). “Immutability and Encapsulation for Sound OO Information Flow Control”. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. DOI: [10.1145/3573270](https://doi.org/10.1145/3573270).
- Runge, T., T. Thüm, L. Cleophas, I. Schaefer, and B. W. Watson (2019b). “Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study”. In: *Formal Methods. FM 2019*

- International Workshops. Refine*. Vol. 12233. Lecture Notes in Computer Science. Springer, pp. 388–405. DOI: [10.1007/978-3-030-54997-8\\_25](https://doi.org/10.1007/978-3-030-54997-8_25).
- Russo, A. and A. Sabelfeld (2010). “Dynamic vs. Static Flow-Sensitive Security Analysis”. In: *Computer Security Foundations Symposium (CSF)*. IEEE, pp. 186–199.
- Sabelfeld, A. and A. C. Myers (2003). “Language-Based Information-Flow Security”. *IEEE Journal on Selected Areas in Communications* 21.1, pp. 5–19.
- Sametinger, J. (1997). *Software Engineering with Reusable Components*. Springer Science & Business Media.
- Schaefer, I., T. Runge, L. Cleophas, and B. W. Watson (2021). “Tutorial: The Correctness-by-Construction Approach to Programming Using CorC”. In: *IEEE Secure Development Conference, SecDev 2021*. IEEE, pp. 1–2. DOI: [10.1109/SecDev51306.2021.00012](https://doi.org/10.1109/SecDev51306.2021.00012).
- Schumann, J. M. (2001). *Automated Theorem Proving in Software Engineering*. Springer Science & Business Media.
- Sery, O., G. Fedyukovich, and N. Sharygina (2012). “Interpolation-Based Function Summaries in Bounded Model Checking”. In: *Hardware and Software: Verification and Testing*. Vol. 7261. Lecture Notes in Computer Science. Springer, pp. 160–175.
- Simonet, V. (2003). “Flow Caml in a Nutshell”. In: *Proceedings of the first APPSEM-II Workshop*, pp. 152–165.
- Smith, C. and S. Drossopoulou (2005). “Chai: Traits for Java-Like Languages”. In: *European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 453–478.
- Sozeau, M. and N. Oury (2008). “First-Class Type Classes”. In: *International Conference on Theorem Proving in Higher Order Logics*. Vol. 5170. Lecture Notes in Computer Science. Springer, pp. 278–293.
- Stickel, M., R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood (1994). “Deductive Composition of Astronomical Software from Subroutine Libraries”. In: *International Conference on Automated Deduction*. Vol. 814. Lecture Notes in Computer Science. Springer, pp. 341–355.
- Strecker, M. (2003). “Formal Analysis of an Information Flow Type System for MicroJava”. *Technische Universität München, Tech. Rep.*
- Sun, Q., A. Banerjee, and D. A. Naumann (2004). “Modular and Constraint-Based Information Flow Inference for an Object-Oriented Language”. In: *International Static Analysis Symposium*. Vol. 3148. Lecture Notes in Computer Science. Springer, pp. 84–99.
- Szyperski, C., D. Gruntz, and S. Murer (2002). *Component Software: Beyond Object-Oriented Programming*. Pearson Education.
- Terauchi, T. and A. Aiken (2005). “Secure Information Flow as a Safety Problem”. In: *International Static Analysis Symposium*. Vol. 3672. Lecture Notes in Computer Science. Springer, pp. 352–367.



- Thüm, T., I. Schaefer, S. Apel, and M. Hentschel (2012). “Family-Based Deductive Verification of Software Product Lines”. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. GPCE '12. Dresden, Germany: Association for Computing Machinery, pp. 11–20.
- Tschannen, J., C. A. Furia, M. Nordio, and N. Polikarpova (2015). “AutoProof: Auto-Active Functional Verification of Object-Oriented Programs”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 9035. Lecture Notes in Computer Science. Springer, pp. 566–580.
- Vassena, M., A. Russo, D. Garg, V. Rajani, and D. Stefan (2019). “From Fine-to Coarse-Grained Dynamic Information Flow Control and Back”. *Proceedings of the ACM on Programming Languages* 3. Proceedings of the ACM on Programming Languages, pp. 1–31.
- Volpano, D. M., C. E. Irvine, and G. Smith (1996). “A Sound Type System for Secure Flow Analysis”. *JCS* 4.2/3, pp. 167–188.
- Watson, B. W., D. G. Kourie, I. Schaefer, and L. Cleophas (2016). “Correctness-by-Construction and Post-hoc Verification: A Marriage of Convenience?” In: *International Symposium on Leveraging Applications of Formal Methods*. Vol. 9952. Lecture Notes in Computer Science. Springer, pp. 730–748.
- Weidenbach, C., D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischniewski (2009). “SPASS Version 3.5”. In: *Proc. of the Intl. Conference on Automated Deduction (CADE)*. Springer, pp. 140–145.
- Wirth, N. (1971). “Program Development by Stepwise Refinement”. *Communications of the ACM* 14.4, pp. 221–227. DOI: [10.1145/362575.362577](https://doi.org/10.1145/362575.362577).
- Wohlin, C., P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén (2012). *Experimentation in Software Engineering*. Springer Science & Business Media.
- Xiang, J. and S. Chong (2021). “Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 18–35.
- Zeyda, F., M. V. M. Oliveira, and A. Cavalcanti (2009). “Supporting ArcAngel in ProofPower”. *Electronic Notes in Theoretical Computer Science* 259, pp. 225–243.
- Zhang, D., Y. Wang, G. E. Suh, and A. C. Myers (2015). “A Hardware Design Language for Timing-Sensitive Information-Flow Security”. *Acm Sigplan Notices* 50.4, pp. 503–516.



# A. Papers of the Thesis

The publications that are part of this cumulative thesis are included in the following order.

1. T. Runge, I. Schaefer, L. Cleophas, T. Thüm, D. Kourie, and B. W. Watson [2019a]. “Tool Support for Correctness-by-Construction”. In: *International Conference on Fundamental Approaches to Software Engineering*. Vol. 11424. Lecture Notes in Computer Science. Springer, pp. 25–42. DOI: [10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2)
2. T. Runge, T. Thüm, L. Cleophas, I. Schaefer, and B. W. Watson [2019b]. “Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study”. In: *Formal Methods. FM 2019 International Workshops. Refine*. Vol. 12233. Lecture Notes in Computer Science. Springer, pp. 388–405. DOI: [10.1007/978-3-030-54997-8\\_25](https://doi.org/10.1007/978-3-030-54997-8_25)
3. T. Runge, T. Bordis, T. Thüm, and I. Schaefer [2021]. “Teaching Correctness-by-Construction and Post-hoc Verification–The Online Experience”. In: *Formal Methods Teaching Workshop*. Vol. 13122. Lecture Notes in Computer Science. Springer, pp. 101–116. DOI: [10.1007/978-3-030-91550-6\\_8](https://doi.org/10.1007/978-3-030-91550-6_8)
4. T. Runge, A. Potanin, T. Thüm, and I. Schaefer [2022b]. “Traits: Correctness-by-Construction for Free”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Vol. 13273. Lecture Notes in Computer Science. Springer, pp. 131–150. DOI: [10.1007/978-3-031-08679-3\\_9](https://doi.org/10.1007/978-3-031-08679-3_9)
5. T. Runge, T. Bordis, A. Potanin, T. Thüm, and I. Schaefer [2023]. “Flexible Correct-by-Construction Programming”. *Logical Methods in Computer Science*
6. T. Runge, M. Servetto, A. Potanin, and I. Schaefer [2022c]. “Immutability and Encapsulation for Sound OO Information Flow Control”. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. DOI: [10.1145/3573270](https://doi.org/10.1145/3573270)
7. T. Runge, A. Knüppel, T. Thüm, and I. Schaefer [2020]. “Lattice-Based Information Flow Control-by-Construction for Security-by-Design”. In: *FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering*. ACM, pp. 44–54. DOI: [10.1145/3372020.3391565](https://doi.org/10.1145/3372020.3391565)
8. T. Runge, A. Kittelmann, M. Servetto, A. Potanin, and I. Schaefer [2022a]. “Information Flow Control-by-Construction for an Object-Oriented Language”. In: *International Conference on Software Engineering and Formal Methods*. Vol. 13550. Lecture Notes in Computer Science. Springer, pp. 209–226. DOI: [10.1007/978-3-031-17108-6\\_13](https://doi.org/10.1007/978-3-031-17108-6_13)

## **A.1. Tool Support for Correctness-by-Construction**



# Tool Support for Correctness-by-Construction

Tobias Runge<sup>1</sup>(✉), Ina Schaefer<sup>1</sup>, Loek Cleophas<sup>2,3</sup>, Thomas Thüm<sup>1</sup>,  
Derrick Kourie<sup>3,4</sup>, and Bruce W. Watson<sup>3,4</sup>

<sup>1</sup> Software Engineering, TU Braunschweig, Braunschweig, Germany  
{tobias.runge,i.schaefer,t.thuem}@tu-bs.de

<sup>2</sup> Software Engineering Technology, TU Eindhoven, Eindhoven, The Netherlands

<sup>3</sup> Information Science, Stellenbosch University, Stellenbosch, South Africa  
{loek,derrick,bruce}@fastar.org

<sup>4</sup> Centre for Artificial Intelligence Research, CSIR, Pretoria, South Africa

**Abstract.** Correctness-by-Construction (CbC) is an approach to incrementally create formally correct programs guided by pre- and postcondition specifications. A program is created using refinement rules that guarantee the resulting implementation is correct with respect to the specification. Although CbC is supposed to lead to code with a low defect rate, it is not prevalent, especially because appropriate tool support is missing. To promote CbC, we provide tool support for CbC-based program development. We present CorC, a graphical and textual IDE to create programs in a simple while-language following the CbC approach. Starting with a specification, our open source tool supports CbC developers in refining a program by a sequence of refinement steps and in verifying the correctness of these refinement steps using the theorem prover KeY. We evaluated the tool with a set of standard examples on CbC where we reveal errors in the provided specification. The evaluation shows that our tool reduces the verification time in comparison to post-hoc verification.

## 1 Introduction

*Correctness-by-Construction* (CbC) [12, 13, 19, 23] is a methodology to construct formally correct programs guided by a specification. CbC can improve program development because every part of the program is designed to meet the corresponding specification. With the CbC approach, source code is incrementally constructed with a low defect rate [19] mainly based on three reasons. First, introducing defects is hard because of the structured reasoning discipline that is enforced by the refinement rules. Second, if defects occur, they can be tracked through the refinement structure of specifications. Third, the trust in the program is increased because the program is developed following a formal process [14].

Despite these benefits, CbC is still not prevalent and not applied for large-scale program development. We argue that one reason for this is missing tool

support for a CbC-style development process. Another issue is that the programmer mindset is often tailored to the prevalent post-hoc verification approach. CbC has been shown to be beneficial even in domains where post-hoc verification is required [29]. In post-hoc verification, a method is verified against pre- and postconditions. In the CbC approach, we refine the method stepwise, and we can check the method partially after each step since every statement is surrounded by a pair of pre- and postconditions. The verification of refinement steps and Hoare triples reduces the proof complexity since the proof task is split into smaller problems. The specifications and code developed using the CbC approach can be used to bootstrap the post-hoc verification process and allow for an easier post-hoc verification as the method constructed using CbC generally is of a structure that is more amenable to verification [29].

In this paper, we present CorC,<sup>1</sup> a tool designed to develop programs following the CbC approach. We deliberately built our tool on the well-known post-hoc verifier KeY [4] to profit from the KeY ecosystem and future extensions of the verifier. We also add CbC as another application area to KeY, which opens the possibility for KeY users to adopt the CbC approach. This could spread the constructive CbC approach to areas where post-hoc verification is prevalent.

Our tool CorC offers a hybrid textual-graphical editor to develop programs using CbC. The textual editor resembles a normal programming editor, but is enriched with support for pre- and postcondition specifications. The graphical editor visualizes the code, its specification, and the program refinements in a tree-like structure. The developers can switch back and forth between both views. In order to support the correct application of the refinement rules, the tool is integrated with KeY [4] such that proof obligations can be immediately discharged during program development. In a preliminary evaluation, we found benefits of CorC compared to paper-and-pencil-based application of CbC and compared to post-hoc verification.

## 2 Foundations of Correctness-by-Construction

Classically, CbC [19] starts with the specification of a program as a Hoare triple comprising a precondition, an abstract statement, and a postcondition. Such a triple, say  $T$ , should be read as a total correctness assertion: if  $T$  is in a state where the precondition holds and its abstract statement is executed, then the execution will terminate and the postcondition will hold.  $T$  will be true for a certain set of concrete program instantiations of the abstract program and false for other instantiations. A refinement of  $T$  is a triple, say  $T'$ , which is true for a subset of concrete programs that render  $T$  to be true.

In our work, pre-/post-condition specifications for programs are written in *first-order logic* (FOL). A formula in FOL consists of atomic formulas which are logically connected. An atomic formula is a predicate which evaluates to true or

---

<sup>1</sup> <https://github.com/TUBS-ISF/CorC>, CorC is an acronym for Correctness-by-Construction.

$\{P\} S \{Q\}$	<i>can be refined to</i>
1. <i>Skip</i> :	$\{P\} \text{ skip } \{Q\}$ iff $P$ implies $Q$
2. <i>Assignment</i> :	$\{P\} x := E \{Q\}$ iff $P$ implies $Q[x := E]$
3. <i>Composition</i> :	$\{P\} S1 ; S2 \{Q\}$ iff there is an intermediate condition $M$ such that $\{P\} S1 \{M\}$ and $\{M\} S2 \{Q\}$
4. <i>Selection</i> :	$\{P\} \text{ if } G_1 \rightarrow S_1 \text{ elseif } \dots G_n \rightarrow S_n \text{ fi } \{Q\}$ iff ( $P$ implies $G_1 \vee G_2 \vee \dots \vee G_n$ ) and $\{P \wedge G_i\} S_i \{Q\}$ holds for all $i$ .
5. <i>Repetition</i> :	$\{P\} \text{ do } [I, V] G \rightarrow S \text{ od } \{Q\}$ iff ( $P$ implies $I$ ) and $(I \wedge \neg G$ implies $Q)$ and $\{I \wedge G\} S \{I\}$ and $\{I \wedge G \wedge V=V_0\} S \{I \wedge 0 \leq V \wedge V < V_0\}$
6. <i>Weaken pre</i> :	$\{P'\} S \{Q\}$ iff $P$ implies $P'$
7. <i>Strengthen post</i> :	$\{P\} S \{Q'\}$ iff $Q'$ implies $Q$
8. <i>Subroutine</i> :	$\{P\} \text{ Sub } \{Q\}$ with subroutine $\{P'\} \text{ Sub } \{Q'\}$ iff $P$ is equal to $P'$ and $Q'$ is equal to $Q$

**Fig. 1.** Refinement rules in CbC [19]

false. Programs in this work are written in the CorC language, which is inspired by the *Guarded Command Language* (GCL) [11] and presented below.

For the concrete instantiation of conditions and assignments, our tool uses a host language. We decided for Java, but other languages are also possible.

To create programs using CbC, we use refinement rules. A Hoare triple is refined by applying rules, which introduce CorC language statements, so that a concrete program is created. The concrete program obtained by refinement is guaranteed to be correct by construction, provided that the correctness-preserving refinement steps have been accurately applied. In Fig. 1, we present the statements and refinement rules used in CbC and our tool.

*Skip.* A skip or empty statement is a statement that does not alter the state of the program (i.e., it does nothing) [11, 19]. This means a Hoare triple with a skip statement evaluates to true if the precondition implies the postcondition.

*Assignment.* An assignment statement assigns an expression of type  $T$  to a variable, also of type  $T$ . In the tool, we use a Java-like assignment ( $x = y$ ). To refine a Hoare triple  $\{P\} S \{Q\}$  with an assignment statement, the assignment rule is used. This rule replaces the abstract statement  $S$  by an assignment  $\{P\} x = E \{Q\}$  iff  $P$  implies  $Q[x := E]$ .

*Composition.* A composition statement is a statement which splits one abstract statement into two. A Hoare triple  $\{P\} S \{Q\}$  is split to  $\{P\} S1 \{M\}$  and  $\{M\} S2 \{Q\}$  in which  $S$  is refined to  $S1$  and  $S2$ .  $M$  is an intermediate condition which evaluates to true after  $S1$  and before  $S2$  is executed [11].

*Selection.* Selection in our CorC language works as a switch statement. It refines a Hoare triple  $\{P\} S \{Q\}$  to  $\{P\} \text{ if } G_1 \rightarrow S_1 \text{ elseif } \dots G_n \rightarrow S_n \text{ fi } \{Q\}$ . The guards  $G_i$  are evaluated, and the sub-statement  $S_i$  of the *first* satisfied guard is executed.

We use a switch-like statement so that every sub-statement has an associated guard for further reasoning. The selection refinement rule can only be used if the precondition  $P$  implies the disjunction of all guards so that at least one sub-statement could be executed.

*Repetition.* The repetition statement  $\{P\} \text{ do } [I, V] G \rightarrow S \text{ od } \{Q\}$  works like a while loop in other languages. If the loop guard  $G$  evaluates to true, the associated loop statement  $S$  is executed. The repetition statement is specified with an invariant  $I$  and a variant  $V$ . To refine a Hoare triple  $\{P\} S \{Q\}$  with a repetition statement, (1) the precondition  $P$  has to imply the invariant  $I$  of the repetition statement, (2) the conjunction of invariant and the negation of the loop guard  $G$  have to imply the postcondition  $Q$ , and (3) the loop body has to preserve the invariant by showing that  $\{I \wedge G\} S \{I\}$  holds. To verify termination, we have to show that the variant  $V$  monotonically decreases in each loop iteration and has 0 as a lower bound.

*Weaken precondition.* The precondition of a Hoare triple can be weakened if necessary. The weaken precondition rule replaces the precondition  $P$  with a new one  $P'$  only if  $P$  implies  $P'$  [12].

*Strengthen postcondition.* To strengthen a postcondition, the strengthen postcondition rule can be used. A postcondition  $Q$  is replaced by a new one  $Q'$  only if  $Q'$  implies  $Q$  [12].

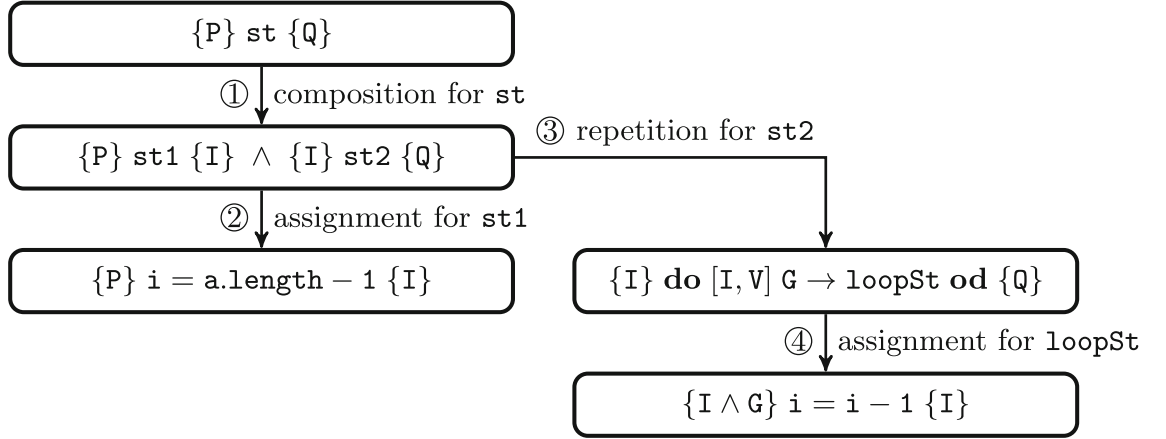
*Subroutine.* A subroutine can be used to split a program into smaller parts. We use a simple subroutine call where we prohibit side effects and parameters. A triple  $\{P\} S \{Q\}$  can be refined to a subroutine  $\{P'\} \text{ Sub } \{Q'\}$ , if the precondition  $P'$  of the subroutine is equal to the precondition  $P$  of the refined statement and the postcondition  $Q'$  of the subroutine is equal to the postcondition  $Q$  of the refined statement. The subroutine can be constructed as a separate CbC program to verify that it satisfies the specification. The Hoare triple  $\{P'\} \text{ Sub } \{Q'\}$  is the starting point to construct a program using CbC.

### 3 Correctness-by-Construction by Example

To introduce the programming style of CbC, we demonstrate the construction of a linear search algorithm using CbC [19]. The linear search problem is defined as follows: We have an integer array  $a$  of some length, and an integer variable  $x$ . We try to find an element in the array  $a$  which has the same value as the variable  $x$ , and we return the index  $i$  where the (last) element  $x$  was found, or  $-1$  if the element is not in the array.

To construct the algorithm, we start with concretizing the pre- and postcondition of the algorithm. Before the algorithm is executed, we know that we have an integer array. Therefore, we specify  $a \neq \text{null} \wedge a.\text{length} \geq 0$  as precondition  $P$ . The postcondition forces that if the index  $i$  is greater than or equal to zero, the element is found on the returned index  $i$  ( $Q := (i \geq 0 \implies a[i] = x)$ ).





**Fig. 2.** Refinement steps for the linear search algorithm

Our algorithm traverses the array in reverse order and checks for each index whether the value is equal to  $x$ . In this case, the index is returned. To create this algorithm, we construct an invariant  $I$  for the loop:

$$I := \neg \text{appears}(a, x, i + 1, a.length) \wedge i \geq -1 \wedge i < a.length$$

The invariant is used to split the array into two parts. A part from  $i + 1$  to  $a.length$  where  $x$  is not contained, and a part from zero to  $i$  which is not checked yet. In every iteration, the next index of the array is checked. The predicate  $\text{appears}(a, x, l, h)$  asserts that  $x$  occurs in array  $a$  inside the range from  $l$  (included) to  $h$  (excluded). The predicate can be translated to FOL as  $\exists i : (i \geq l \wedge i < h \wedge a[i] = x)$ .

We can use the CbC refinement rules to implement linear search. The refinement steps for the example are shown in Fig. 2 and numbered from ① to ④. To create a loop in the program, we need to initialize a loop counter variable to establish the invariant. Therefore, we split the program by introducing a composition statement (① in Fig. 2). The invariant  $I$  is used as intermediate condition (i.e.,  $M := I$ ), because it has to be true after the initialization, and before the first loop step. The statement  $\text{st1}$  is refined to an assignment statement ②. We initialize  $i$  with  $a.length - 1$  to start at the end of the array. This assignment satisfies the intermediate condition  $I$  where  $i$  is replaced by  $a.length - 1$ . The range of  $\text{appears}$  is empty, and therefore the predicate evaluates to true. To refine the second statement ( $\text{st2}$ ), we use the repetition refinement rule ③. As long as  $x$  is not found, we iterate through the array. As guard of the repetition, we use  $(i \geq 0 \wedge a[i] \neq x)$ . The invariant of the repetition is the invariant  $I$  introduced above. The variant  $V$  is  $i + 1$ . To verify that this refinement is valid, we have to verify that the precondition of the repetition statement implies the invariant, and that the invariant and the negated guard imply the postcondition of the repetition (cf. Rule 5). Both are valid because the precondition is equal to the invariant and the postcondition of the repetition statement (in this case it is  $Q$ ) is equal to the negated guard. The last step is to refine the abstract loop statement ( $\text{loopSt}$ ) ④. We use an assignment to decrease  $i$  and get the final

program. We can verify that the invariant holds after each loop iteration. The program terminates because the variant decreases in every step and it is always greater than or equal to zero.

## 4 Tool Support in CorC

CorC extends KeY’s application area by enabling CbC to spread the constructive engineering to areas where post-hoc verification is prevalent. KeY programmers can use both approaches to construct formally correct programs. By using CorC, they develop specification and code that can bootstrap the post-hoc verification. The CorC tool<sup>2</sup> is realized as an Eclipse plug-in in Java. We use the Eclipse Modeling Framework (EMF)<sup>3</sup> to specify a CbC meta model. This meta model is used by two editor views, a textual and a graphical editor. The Hoare triple verification is implemented by the deductive program verification tool KeY [4]. In the following list, we summarize the features of CorC.

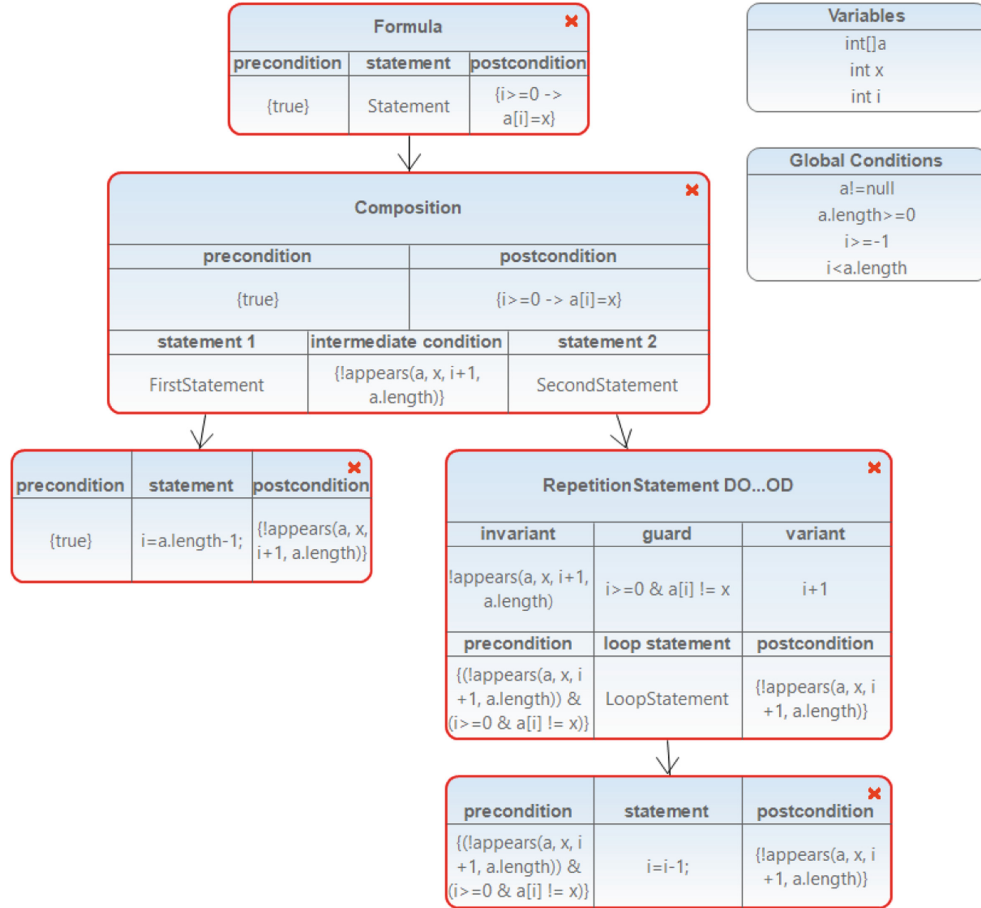
- Programs are written as Hoare triple specifications, including pre-/postcondition specifications and abstract statements or assignment/skip statements in concrete triples.
- CorC has eight rules to construct programs: skip, assignment, composition, selection, repetition, weakening precondition, strengthening postcondition, and subroutine (cf. Sect. 2).
- Pre-/postconditions and invariant specification are automatically propagated through the program.
- CorC comprises a graphical and a textual editor that can be used interchangeably.
- Up to now, CorC supports integers, chars, strings, arrays, and subroutine calls without side effects, I/O, and library calls.
- Hoare triples are typically verified by KeY automatically. If the proof cannot be closed automatically, the user can interact with KeY.
- Helper methods written in Java 1.5 can be used in a specification.
- CorC comprises content assist and an automatic generation of intermediate conditions.

### 4.1 Graphical Editor

The graphical editor represents CbC-based program refinement by a tree structure. A node represents the Hoare triple of a specific CorC language statement. Figure 3 presents the linear search algorithm of Sect. 3 in the graphical editor. The structure of the tree is the same as in Fig. 2. The additional nodes on the right specify used program variables including their type and global invariant

<sup>2</sup> <https://github.com/TUBS-ISF/CorC>.

<sup>3</sup> <https://eclipse.org/emf/>.



**Fig. 3.** Linear search example in the graphical editor

conditions. The global invariant conditions are added to every pre- and postcondition of Hoare triples to simplify the construction of the program. In the example, we specify the array  $a$  and the range of variable  $i$  to support the verification, as KeY requires this range to be explicit for verification.

The root node of the tree shows the abstract Hoare triple for the overall program with a symbolic name for the abstract statement. In every node, the pre- and postcondition are specified on the left and right of the node under the corresponding header. A composition statement node, the second statement of the tree, contains the pre- and postcondition and additionally defines an intermediate condition. The intermediate condition is the middle term in the bottom line. Both abstract sub-statements of the composition have a symbolic name and can be further refined by adding a connection to another node (i.e., creating a parent-child relation). The repetition node contains fields to specify the invariant, the guard and the variant of the repetition. These fields are in the middle row. The pre- and postcondition are associated to the inner loop statement. An assignment node (cf. both leaf nodes of the figure) contains the precondition, the assignment, and the postcondition. The representations of the nodes for the refinements not illustrated in this example are similar.

Refinement steps are represented by edges. The pre- and postconditions are propagated from parents to their children on drawing the parent/child relation. We explicitly show the propagated conditions in a node to improve readability. The propagated conditions from the parent are unmodifiable because refinement rules determine explicitly how conditions are propagated. An exception are the rules to weaken the precondition or strengthen the postcondition. Here, the conditions can be overridden. At the repetition statement, we only depict the pre-/postconditions of the inner loop statement to reduce the size of this node. The pre-/postconditions of the parent node (in our example the composition statement) are not shown explicitly, but they are propagated internally to verify that the repetition refinement rule is satisfied. To visualize the verification status, the nodes have a green border if proven, a red one otherwise.

By showing the Hoare triples explicitly, problems in the program can be localized. If some leaf node cannot be proven, the user has to check the assignment and the corresponding pre-/postcondition. If an error occurred, the conditions on the refinement path up to pre-/postcondition of the starting Hoare triple can be altered. Other paths do not need to be checked. To prove the program correct, we have to prove that the refinement is correct. Aside from the side conditions of refinement rules (cf. iff conditions in refinement rules), only the leaf nodes of the refinement tree which contain basic Hoare triples with skip or assignment statements need to be verified by a prover, while all composite statements are correct by construction of their conditions.

To support the user in developing intermediate conditions for composition statements, our tool can compute the weakest precondition from a postcondition and a concrete assignment by using the KeY theorem prover. So, the user can create a specific assignment statement and generate the intermediate conditions afterwards. We also support modularization, to cover cases where algorithms become too large. Sub-algorithms can be created using CbC in other CorC programs. We introduce a simple subroutine rule which can be used as a leaf node in the editor. The subroutine has a name and it is connected to a second diagram with the same name as the subroutine. This subroutine call is similar to a classic method call. It can be used to decompose larger CbC developments to multiple smaller programs.

## 4.2 Textual Editor

The textual editor is an editor for the CorC programming language described above. The user writes code by using keywords for the specific statements and enriches the code with conditions, such as invariants or intermediate conditions, and assignments in our CorC syntax. The syntax of the composed statements in the textual editor is shown in Fig. 4. In the `GlobalConditions` declaration, we enumerate the needed global conditions separated with a comma. The used variables are enumerated after the `JavaVariables` keyword.

The linear search example program presented in Sect. 3 is shown in the syntax of CorC in Listing 1. The program starts with keyword `Formula`. The pre- and postcondition of the abstract Hoare triple are written after the `pre:` and `post:`

<i>Selection statement</i>	<i>Repetition statement</i>
<code>if ("guard") then {statement}</code>	<code>while ("guard")</code>
<code>elseif ("guard") then {statement}</code>	<code>inv: ["invariant"] var: ["variant"]</code>
<code>...</code>	<code>do {statement} od</code>
<code>fi</code>	

**Fig. 4.** Syntax of statements in textual editor

```

1  Formula "linearSearch"
2  pre: {"true"}
3  {
4    {
5      i=a.length-1;
6    }
7    intm: ["!appears(a, x, i+1, a.length)"]
8    {
9      while ("i>=0 & a[i]!=x")
10     inv: ["!appears(a, x, i+1, a.length)"]
11     var: ["i+1"] do
12     {
13       i=i-1;
14     } od
15   }
16 }
17 post: {"i>=0 -> a[i]=x"}
18
19 GlobalConditions
20   conditions {"a!=null", "a.length>=0",
21             "i>=-1", "i<a.length"}
22
23 JavaVariables
24   variables {"int[] a", "int x", "int i"}

```

**Listing 1.** Linear search example in the textual editor

keywords. The abstract statement of the Hoare triple is refined to a composition statement in lines 3–16. The statements are surrounded by curly brackets to establish the refinement structure. We have the first statement in lines 4–6, the intermediate condition in line 7 and the second statement in lines 8–15. The first statement is refined to an assignment (Line 5). The refinement is done by introducing an assignment in Java syntax (`i = a.length - 1`);. The second statement is refined to a repetition statement (cf. the syntax of a repetition statement in Fig. 4). We specify the guard, the invariant, and the variant. Finally, the single statement of the loop body is refined to an assignment in Line 13.

As in the graphical editor, pre-/postconditions are propagated top-down from a parent to a child statement. For example, the intermediate condition of a

```

1  \javaSource "src";
2  \include "helper.key";
3  \programVariables {int x;}
4  \problem {
5    (x = 0) -> \<{x=x+1;}> (x = 1)
6  }

```

Listing 2. KeY problem file

composition statement which is the postcondition of the first sub-statement and the precondition of the second, appears only once in the editor (e.g., Line 7). To support the user, we implemented syntax highlighting and a content assist. When starting to write a statement, a user may employ auto-completion where the statements are inserted following the syntax in Fig. 4. The user can specify the conditions, then the next statement can be refined. The editor also automatically checks the syntax and highlights syntax errors. Information markers are used to indicate statements which are not proven yet. For example, the Hoare triple of the assignment statement (`i = a.length - 1`) in Listing 1 has to be verified, and CorC marks the statement according to the proof completion results.

### 4.3 Verification of CorC Programs

To prove the refined program is correct, we have to prove side conditions of refinements correct (e.g., prove that an assignment satisfies the pre-/postcondition specification). This reduces the proof complexity because the challenge to prove a complete program is decomposed into smaller verification tasks. The intermediate Hoare triples are verified indirectly through the soundness of the refinement rules and the propagation of the specifications from parent nodes to child nodes [19]. Side conditions occur in all refinements (cf. iff conditions in refinement rules). These side conditions, such as the termination of repetition statements or that at least one guard in a selection has to evaluate to true, are proven in separate KeY files.

For the proof of concrete Hoare triples, we use the deductive program verifier KeY [4]. Hoare triples are transformed to KeY's dynamic logic syntax. The syntax of KeY problem files is shown in Listing 2. Using the keyword `javaSource`, we specify the path to Java helper methods which are called in the specifications. These methods have to be verified independently with KeY. A KeY helper file, where the users can define their own FOL predicates for the specification, is included with the keyword `include`. For example, in CorC a predicate *appears*( $a, x, l, h$ ) (cf. the linear search example) can be used which is specified in the helper file as a FOL formula. The variables used in the program are listed after the keyword `programVariables`. After `problem`, we define the Hoare triple to be proven, which is translated to dynamic logic as used by KeY. KeY problem files are verified by KeY. As we are only verifying simple Hoare triples with skip

or assignment statements, KeY is usually able to close the proofs automatically if the Hoare triple is valid.

To verify total correctness of the program, we have to prove that all repetition statements terminate. The termination of repetition statements is shown by proving that the variants in the program monotonically decrease and are bounded. Without loss of generality, we assume this bound to equal 0, as this is what KeY requires. This is done by specifying the problem in the KeY file in the following way: `(invariant & guard) -> {var0:=var} \<{std}\> (invariant & var<var0 & var>=0)`. The code of the loop body is specified at `std` to verify that after one iteration of the loop body the variant `var` is smaller than before but greater than or equal to zero.

To verify Hoare triples in the graphical editor, we implemented a menu entry. The user can right-click on a statement and start the automatic proof. If the proof is not closed, the user can interact with the opened KeY interface. To prove Hoare triples in the textual editor, we automatically generate all needed problem files for KeY whenever the user saves the editor file. The proof of the files is started using a menu button. The user gets feedback which triples are not proven by means of markers in the editor.

#### 4.4 Implementation as Eclipse Plugin

We extended the Eclipse modeling framework with plugins to implement the two editors. We have created a meta model of the CbC language to represent the required constructs (i.e., statements with specification). The statements can be nested to create the CbC refinement hierarchy. The graphical and the textual editor are projections on the same meta model. The graphical editor is implemented using the framework Graphiti.<sup>4</sup> It provides functionality to create nodes and to associate them to domain elements, such as statements and specifications. The nodes can be added from a palette at the side of the editor, so no incorrect statement with its associated specification can be created. We implemented editing functionality to change the text in the node; the background model is changed simultaneously. Graphiti also provides the possibility to update nodes (e.g., to propagate pre- and postconditions), if we connect those nodes by refinement edges. The refinement is checked for compliance with the CbC rules.

The textual editor is implemented using XText.<sup>5</sup> We created a grammar covering every statement and the associated specification. If the user writes a program, the text is parsed and translated to an instance of the meta model. If a program is created in one editor, a model (an instance of our meta model) of the program is created in the background. We can easily transform one view into the other. The transformation is a generation step and not a live synchronization between both views, but it is carried out invisibly for the user when changing the views.

---

<sup>4</sup> <https://eclipse.org/graphiti/>.

<sup>5</sup> <https://eclipse.org/Xtext/>.



**Table 1.** Evaluation of the example programs

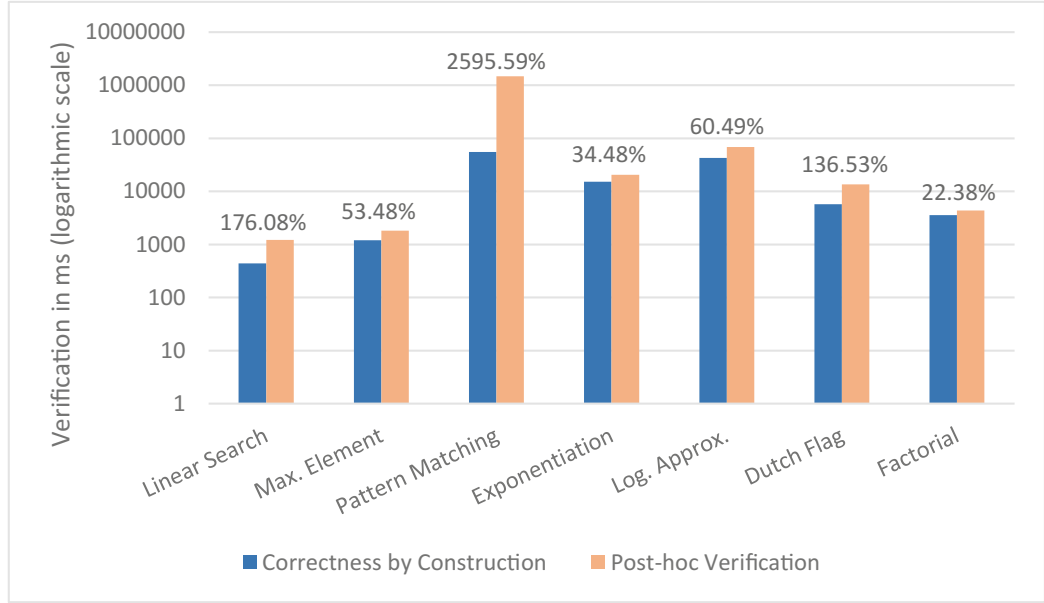
Algo- rithm	#Nodes in GE	#Lines in TE	#Lines with JML	#Verified CorC triples	CbC Total Proof- Nodes	CbC Total Proof- Time	PhV Total Proof- Nodes	PhV Total Proof- Time
Linear Search	5	12	10	5/5	285	0.4 s	589	1.2 s
Max. Element	9	21	15	9/9	1023	1.2 s	993	1.8 s
Pattern Matching	14	23	20	13/13	21131	54.9 s	201619	1479.3 s
Exponen- tiation	7	21	17	7/7	6588	15.2 s	7303	20.4 s
Log. Approx.	5	16	12	5/5	13756	42.7 s	18835	68.5 s
Dutch Flag	8	26	24	8/8	4107	5.7 s	4993	13.4 s
Factorial	5	15	13	4/4	1554	3.6 s	1598	4.4 s

(GE) Graphical Editor, (TE) Textual Editor, (PhV) Post-hoc Verification

In implementing CorC, we considered the exchangeability of the host language. The specifications and assignments are saved as strings in the meta model. They are checked by a parser to comply with Java. This parser could be exchanged to support a different language. The verification is done by generating KeY files which are then evaluated by KeY. Here, we have to exchange the generation of the files if another theorem prover should be integrated. The information of the meta model may have to be adopted to fit the needs of the other prover. We also have to implement a programmatic call to the other prover.

## 5 Evaluation

The tool support offers new chances to evaluate CbC versus post-hoc verification. We quantitatively compare the development and verification of programs with CorC and with post-hoc verification. This is to check the hypothesis that the verification of algorithms is faster with CorC than with post-hoc verification. We created the first eight algorithms from the book by Kourie and Watson [19] in our graphical editor. For comparison purposes, we also wrote each example as a plain Java program with JML specifications in order to directly verify it with KeY. The specifications are the same as in CorC. We measured the verification time and the proof nodes that KeY needed to close the proofs for both approaches. The results of the evaluation are presented in Table 1 (verification time rounded).



**Fig. 5.** Proof time of CbC and post-hoc verification in logarithmic scale

The algorithms have 5 to 14 nodes in the graphical editor and 12 to 26 lines of code in the textual editor. The Java version with a JML specification always has fewer lines (between 8% and 29% smaller). The additional specifications, such as the intermediate conditions of composition statements, and the global invariant conditions and variables cause more lines of code in the CbC program.

The verification of the eight algorithms worked nearly without problems. We verified 7 out of 8 examples within CorC. In the cases without problems, every Hoare triple and the termination of the loops could be proven. We had to prove fewer Hoare triples than nodes in the editor, as not every node has to be proven separately. Composition nodes are proven indirectly through the refinement structure. For *exponentiation*, *logarithm*, and *factorial*, we had to implement recursive helper methods which are used in the specification. Therefore, the programs impose upper bounds for integers to shorten the proof. The *binary search* algorithm could not be verified automatically in KeY using post-hoc verification or CorC. In each step, when the element is not found, the algorithm halves the array. KeY could not prove that the searched element is in the new boundaries because verification problems with arithmetic division are hard to prove for KeY automatically.

In the case of measured proof nodes, *maximum element* needs slightly fewer nodes proved with post-hoc verification than with CbC. In the other cases, the proofs for the algorithms constructed with CbC are 3% to 854% smaller. The largest difference was measured for the *pattern matching* algorithm. The proof is reduced to a ninth of the nodes.

The verification time is visualized in Fig. 5. The time is measured in milliseconds and scaled logarithmically. The proofs for the CbC approach are always faster showing lower proof complexity. For *maximum element*, *exponentiation*,

*logarithm* and *factorial*, the post-hoc verification time requires between 22% and 60% more time. The difference increases for *Dutch flag* and *linear search* to 137% and 176%, respectively. Algorithm *pattern matching* has the biggest difference. Here, the CbC approach needs nearly a minute, but the post-hoc approach needs over 24 min. To verify our hypothesis, we apply the non-parametric paired Wilcoxon-Test [30] with a significance level of 5%. We can reject the null hypothesis that CbC verification and post-hoc verification have no significant difference in verification time (p-value = 0.007813). This rejection of the null hypothesis is an empirical evidence for our hypothesis that verification is faster with CorC than with post-hoc verification.

With our tool support, we were able to compare the CbC approach with post-hoc verification. For our examples, we evaluated that the verification effort is reduced significantly which indicates a reduced proof complexity. It is worthwhile to further investigate the CbC approach, also to profit from synergistic effects in combination with post-hoc verification. As we built CorC on top of KeY, the post-hoc verification of programs constructed with CorC is feasible.

An advantage of CorC is the overview on all Hoare triples during development. In this way, we found some specifications where descriptions in the book by Kourie and Watson [19] were not precise enough to verify the problem in KeY. For example, in the *pattern matching* algorithm, we had to verify two nested loops. At one point, we had to verify that the invariant of the inner loop implies the invariant of the outer loop. This was not possible, so we extended the invariant of the inner loop to be the conjunction of both invariants. In the book of Kourie and Watson [19], this conjunction of both invariants was not explicitly used.

## 6 Related Work

We compare CorC to other programming languages and tools using specification or refinements. The programming language Eiffel is an object-oriented programming language with a focus on design-by-contract [21, 22]. Classes and methods are annotated with pre-/postconditions and invariants. Programs written in Eiffel can be verified using AutoProof [18, 28]. The verification tool translates the program with assertions to a logic formula. An SMT-solver proves the correctness and returns the result. Spec# is a similar tool for specifying C# programs with pre-/postcondition contracts. These programs can be verified using Boogie. The code and specification is translated to an intermediate language (BoogiePL) and verified [5, 6]. VCC [8] is a tool to annotate and verify C code. For this purpose, it reuses the Spec# tool chain. VeriFast [16] is another tool to verify C and Java programs with the help of contracts. The contracts are written in separation logic (a variant of Hoare logic). As in Eiffel, the focus of Spec#, VCC, and VeriFast is on post-hoc verification and debugging failed proof attempts.

The Event-B framework [2] is a related CbC approach. Automata-based systems including a specification are refined to a concrete implementation.

Atelier B [1] implements the B method by providing an automatic and interactive prover. Rodin [3] is another tool implementing the Event-B method. The main difference to CorC is that CorC works on code and specifications rather than on automata-based systems.

ArcAngel [25] is a tool supporting Morgan’s refinement calculus. Rules are applied to an initial specification to produce a correct implementation. The tool implements a tactic language for refinements to apply a sequence of rules. In comparison to our tool, ArcAngel does not offer a graphical editor to visualize the refinement steps. Another difference is that ArcAngel creates a list of proof obligations which have to be proven separately. CRefine [26] is a related tool for the Circus refinement calculus, a calculus for state-rich reactive systems. Like our tool, CRefine provides a GUI for the refinement process. The difference is that we specify and implement source code, but they use a state-based language. ArcAngelC [10] is an extension to CRefine which adds refinement tactics.

The tools iContract [20] and OpenJML [9] apply design-by-contract. They use a special comment tag to insert conditions into Java code. These conditions are translated to assertions and checked at runtime which is a difference to our tool because no formal verification is done. DBC-Python is a similar approach for the Python language which also checks assertions at runtime [27].

To verify the CbC program, we need a theorem prover for Hoare triples, such as KeY [4]. There are other theorem provers which could be used (e.g., Coq [7] or Isabelle/HOL [24]). The Tecton Proof System [17] is a related tool to structure and interactively prove Hoare logic specification. The proofs are represented graphically as a set of linked trees. These interactive provers do not fit our needs because we want to automate the verification process. KeY provides a symbolic execution debugger (SED) that represents all execution paths with specifications of the code to the verification [15]. This visualization is similar to our tree representation of the graphical editor. The SED can be used to debug a program if an error occur during the post-hoc verification process.

## 7 Conclusion and Future Work

We implemented CorC to support the Correctness-by-Construction process of program development. We created a textual and a graphical editor that can be used interchangeably to enable different styles of CbC-based program development. The program and its specification are written in one of the editors and can be verified using KeY. This reduces the proof complexity with respect to post-hoc verification. We extended the KeY ecosystem with CorC. CorC opens the possibility to utilize CbC in areas where post-hoc verification is used as programmers could benefit from synergistic effects of both approaches. With tool support, CbC can be studied in experiments to determine the value of using CbC in industry.

For future work, we want to extend the tool support, and we want to evaluate empirically the benefits and drawbacks of CorC. To extend the expressiveness, we implement a rule for methods to use method calls in CorC. These methods have to be verified independently by CorC/KeY. We could investigate whether the method call rules of KeY can be used for our CbC approach. Another future work is the inference of conditions to reduce the manual effort. Postconditions can be generated automatically for known statements by using the strongest postcondition calculus. Invariants could be generated by incorporating external tools. As mentioned earlier, other host languages and other theorem provers can be integrated in our IDE.

The second work package for future work comprise the evaluation with a user study. We could compare the effort of creating and verifying algorithms with post-hoc verification and with our tool support. The feedback can be used to improve the usability of the tool.

## References

1. Abrial, J.R.: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, Cambridge (2005)
2. Abrial, J.R.: *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transfer* **12**(6), 447–466 (2010)
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: *Deductive Software Verification - The KeY Book: From Theory to Practice*, vol. 10001. Springer, Heidelberg (2016). <https://doi.org/10.1007/978-3-319-49812-6>
5. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. *Commun. ACM* **54**(6), 81–91 (2011)
6. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) *CASSIS 2004*. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-30569-9\\_3](https://doi.org/10.1007/978-3-540-30569-9_3)
7. Bertot, Y., Castéran, P.: *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-662-07964-5>
8. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03359-9\\_2](https://doi.org/10.1007/978-3-642-03359-9_2)
9. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) *NFM 2011*. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20398-5\\_35](https://doi.org/10.1007/978-3-642-20398-5_35)

10. Conserva Filho, M., Oliveira, M.V.M.: Implementing tactics of refinement in *CRefine*. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) *SEFM 2012*. LNCS, vol. 7504, pp. 342–351. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33826-7\\_24](https://doi.org/10.1007/978-3-642-33826-7_24)
11. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* **18**(8), 453–457 (1975)
12. Dijkstra, E.W.: *A Discipline of Programming*. Prentice Hall, Upper Saddle River (1976)
13. Gries, D.: *The Science of Programming*. Springer, Heidelberg (1987). <https://doi.org/10.1007/978-1-4612-5983-1>
14. Hall, A., Chapman, R.: Correctness by construction: developing a commercial secure system. *IEEE Softw.* **19**(1), 18–25 (2002)
15. Hentschel, M.: Integrating symbolic execution, debugging and verification. Ph.D. thesis, Technische Universität Darmstadt (2016)
16. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the verifast program verifier. In: Ueda, K. (ed.) *APLAS 2010*. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-17164-2\\_21](https://doi.org/10.1007/978-3-642-17164-2_21)
17. Kapur, D., Nie, X., Musser, D.R.: An overview of the Tecton proof system. *Theoret. Comput. Sci.* **133**(2), 307–339 (1994)
18. Khazeev, M., Rivera, V., Mazzara, M., Johard, L.: Initial steps towards assessing the usability of a verification tool. In: Ciancarini, P., Litvinov, S., Messina, A., Sillitti, A., Succi, G. (eds.) *SEDA 2016*. AISC, vol. 717, pp. 31–40. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-70578-1\\_4](https://doi.org/10.1007/978-3-319-70578-1_4)
19. Kourie, D.G., Watson, B.W.: *The Correctness-by-Construction Approach to Programming*. Springer, Heidelberg (2012). <https://doi.org/10.1007/978-3-642-27919-5>
20. Kramer, R.: *iContract - the Java design by contract tool*. In: *Proceedings, Technology of Object-Oriented Languages. TOOLS 26 (Cat. No. 98EX176)*, pp. 295–307. IEEE, August 1998
21. Meyer, B.: Eiffel: a language and environment for software engineering. *J. Syst. Softw.* **8**(3), 199–246 (1988)
22. Meyer, B.: Applying “design by contract”. *Computer* **25**(10), 40–51 (1992)
23. Morgan, C.: *Programming from Specifications*, 2nd edn. Prentice Hall, Upper Saddle River (1994)
24. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): *Isabelle/HOL*. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
25. Oliveira, M.V.M., Cavalcanti, A., Woodcock, J.: ArcAngel: a tactic language for refinement. *Formal Aspects Comput.* **15**(1), 28–47 (2003)
26. Oliveira, M.V.M., Gurgel, A.C., Castro, C.G.: CRefine: support for the circus refinement calculus. In: *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pp. 281–290. IEEE, November 2008
27. Plosch, R.: Tool support for design by contract. In: *Proceedings, Technology of Object-Oriented Languages. TOOLS 26 (Cat. No. 98EX176)*, pp. 282–294. IEEE, August 1998
28. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) *TACAS 2015*. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46681-0\\_53](https://doi.org/10.1007/978-3-662-46681-0_53)



29. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.: Correctness-by-construction and post-hoc verification: a marriage of convenience? In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 730–748. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-47166-2\\_52](https://doi.org/10.1007/978-3-319-47166-2_52)
30. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Heidelberg (2012). <https://doi.org/10.1007/978-3-642-29044-2>

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





## **A.2. Comparing Correctness-by-Construction with Post-Hoc Verification — A Qualitative User Study**



# Comparing Correctness-by-Construction with Post-Hoc Verification—A Qualitative User Study

Tobias Runge<sup>1(✉)</sup>, Thomas Thüm<sup>2(✉)</sup>, Loek Cleophas<sup>3,4(✉)</sup>, Ina Schaefer<sup>1(✉)</sup>,  
and Bruce W. Watson<sup>4,5(✉)</sup>

<sup>1</sup> TU Braunschweig, Braunschweig, Germany

{tobias.runge,i.schaefer}@tu-bs.de

<sup>2</sup> University of Ulm, Ulm, Germany

thomas.thuem@uni-ulm.de

<sup>3</sup> TU Eindhoven, Eindhoven, The Netherlands

loek@fastar.org

<sup>4</sup> Stellenbosch University, Stellenbosch, South Africa

bruce@fastar.org

<sup>5</sup> Centre for Artificial Intelligence Research, Stellenbosch, South Africa

**Abstract.** Correctness-by-construction (CbC) is a refinement-based methodology to incrementally create formally correct programs. Programs are constructed using refinement rules which guarantee that the resulting implementation is correct with respect to a pre-/postcondition specification. In contrast, with post-hoc verification (PhV) a specification and a program are created, and afterwards verified that the program satisfies the specification. In the literature, both methods are discussed with specific advantages and disadvantages. By letting participants construct and verify programs using CbC and PhV in a controlled experiment, we analyzed the claims in the literature. We evaluated defects in intermediate code snapshots and discovered a trial-and-error construction process to alter code and specification. The participants appreciated the good feedback of CbC and state that CbC is better than PhV in helping to find defects. Nevertheless, some defects in the constructed programs with CbC indicate that the participants need more time to adapt the CbC process.

## 1 Introduction

*Correctness-by-construction* (CbC) [17, 19, 25, 30] as proposed by Dijkstra is a method for the construction of formally correct programs. The programmer refines an abstract statement with pre-/postcondition specification to a concrete implementation, guided by the specification and refinement rules. It is claimed that programmers construct programs with low defect rates with CbC [20]. There are three reasons for this that need to be evaluated. First, the structured reasoning discipline which is enforced by the refinement rules reduces the possibility to introduce

defects. Second, defects in the code can be traced to their source through the refinement structure. Third, programmers and users gain trust in the program because a formal methodology was used to create the program [25]. We implemented the correctness-by-construction approach in a graphical IDE called CorC,<sup>1</sup> which support users during the construction and verification of programs.

With deductive post-hoc verification (PhV), we refer to techniques as used in the KeY community [4], which verify a program after its creation. A verifier checks whether the program satisfies its pre-/postcondition specification. PhV does not provide a strict guideline on how to construct the program; the programmer can freely implement the program. This can decrease the time taken to create a first (potentially faulty) version of a program, but can increase the program verification time because it is more likely that defects occur in the code [36]. In order to evaluate this claim, we consider the post-hoc verifier KeY [4] as an instance. KeY can verify Java programs annotated with pre-/postcondition specifications in the Java Modeling Language (JML).

As the title suggests, we compare correctness-by-construction with post-hoc verification. In a qualitative user study, participants use CorC and KeY to implement and verify an algorithm with each tool. By analyzing 347 intermediate code snapshots, we get better insights in the process used by participants to construct and verify algorithms. With a user experience questionnaire, we compare which advantages and disadvantages of the verification techniques and the tools have been experienced. Our contributions in this paper are the following.

- We give an overview of advantages and disadvantages of CbC and PhV.
- We designed and performed a user study to compare both approaches. We analyze the defects in code and specification of each intermediate snapshot for both tools.
- We discuss our insights and compare CbC with PhV based on our user study.

## 2 Verification Techniques

In our user study, we evaluate the techniques PhV and CbC. Therefore, we first present and compare the foundations of both techniques. We also survey claims about their advantages and disadvantages as discussed in the research literature.

### 2.1 Post-hoc Verification

With post-hoc verification, we refer to a method which is used to verify whether a program satisfies a given specification. A programmer develops a program and a pre-/postcondition specification. Besides the pre-/postcondition specification, loop invariants can be defined to specify the behavior of loops in the code. The correctness of the program can be verified by using a deductive verification tool, such as KeY [4]. It translates the program and the specification to a dynamic logic formula (i.e., proof obligations). The program is executed symbolically, and

<sup>1</sup> see <https://github.com/TUBS-ISF/CorC> and [34] for explanation of the editor.

the formula is updated according to the new symbolic state. After the program is completely executed, it no longer appears in the formula, and the remaining first-order proof goal can be evaluated by theorem proving. The verification can be performed (semi-)automatically or interactively. We use automatic verification in this paper in order to be able to focus the user study on the construction of programs and specification. Most users in industry do not have a theoretical background to verify programs interactively.

## 2.2 Correctness-by-Construction

Correctness-by-construction in the classical Dijkstra-style [17, 25] is a programming method which starts with a Hoare triple specification. This Hoare triple contains a precondition, an abstract statement (i.e., a statement that is a placeholder for concrete code), and a postcondition. The triple asserts total correctness. If the program is in a state where the precondition holds, its execution will terminate in a state where the postcondition holds. An abstract statement in a Hoare triple can be refined to a concrete program using refinement rules. The rules introduce new statements, such as loops or assignments. By refining the program, the pre-/postcondition specification is propagated through the constructed program, so that the refined statements are also surrounded by a pre- and a postcondition, forming more Hoare triples [17, 25]. These refinement rules introduce proof obligations which have to be discharged to establish the correct application of the refinements rules. E.g., it has to be verified that by executing an assignment the corresponding postcondition is implied, or that a loop invariant holds after each iteration. The correctness of these proof obligations can be checked using verification tools [1, 34]. We implemented tool support for the construction of programs following CbC [34]. The graphical editor CorC visualizes program refinements in a tree-like structure.

## 2.3 Contrasting Correctness-by-Construction and Post-hoc Verification

CbC and PhV are two different methods to create verified software. Nevertheless, they share commonalities. Both start with a pre-/postcondition specification and result in a program that satisfies this specification. The procedure to construct the program, however, is different. With CbC, the program is constructed stepwise by applying checkable refinement rules. With PhV, the program is constructed without a strict guideline (i.e., the programmer can freely develop the program and intermediate steps are not proven). Afterwards, the final program can be verified.

It is claimed that CbC can lead to well-structured code that can be verified more easily [25, 36]. The additional time needed to construct the code is said to be amortized with a significantly reduced time to prove the code. When applying CbC, every refined statement leads to a provable side condition, where a theorem prover can check whether this condition is satisfied. If the check fails, the programmer can alter the refined statement to establish the proof. This is

a potential advantage compared to PhV because problems in the verification process can be pinned to small parts of the program. In contrast, with PhV additional expertise or sophisticated tool support is necessary to infer the defect from open goals in the proof [33].

Programmers who use the CbC approach are bound to the stepwise refinement using rules. Therefore, after each refinement the program with all conditions can be reviewed by the programmers. They can continuously check the surrounding specification of every statement. This can raise awareness of defects in the program, resulting in fewer defects in comparison to PhV programming. The number of required iterations to get to a correct program with CbC may also be reduced because defects are detected early, even before a prover is used [36].

An open question is whether the experience of developers is crucial for the development of correct code. Using PhV, programmers can implement algorithms as they normally do and verify whether the program is correct afterwards. Using CbC, the programmer needs an understanding of the refinement rules to construct programs. Whether this barrier noticeably increases the time of the construction process, or whether the CbC method does not have a negative influence needs to be evaluated.

These claims are established in the literature but need to be evaluated in a user study. We analyze defects in intermediate and final programs and interpret the answers of a questionnaire to provide evidence for the claims.

### 3 Design of a User Study

To qualitatively evaluate CbC and PhV, we performed a user study with the two tools, CorC and KeY. We decided explicitly for a controlled experiment to monitor all participants in parallel during the tasks and to collect all programming results. We selected CorC because it is a new tool that supports the CbC method in a graphical user interface and which has been taught to the participants. KeY, which is a major tool for the automatic verification of Java programs, is used to get good comparability as CorC uses KeY as back-end for the verification. Therefore, we have a comparable expressiveness with both tools.

We provide the participants a pre-/postcondition specification for an algorithm, and they developed code to satisfy this specification. The algorithms can be implemented in under ten lines of code. We decided explicitly for this size, so the whole experiment could be done in 90 min because it is complicated to motivate people to do longer experiments. We also excluded the process of writing an adequate pre-/postcondition specification because this has to be done for both techniques and highly influences what needs to be implemented and verified. The same starting point reduces the divergence, so that we can analyze the results on the same basis. We want to qualitatively analyze how the participants develop and verify code. Therefore, we took intermediate snapshots of the code every time the code was verified and analyzed the defects created during the development process. We checked a total of 347 versions of programs, something which is not feasible with larger programs and more participants. The user

experience with the tools was measured qualitatively by a questionnaire in order to find improvement potentials. The material of the user study is published on GitHub.<sup>2</sup>

**Objective.** We surveyed in Sect. 2.3 whether CbC can have a positive impact on programming and verifying code. Hence, we want to evaluate whether a positive impact can be detected (i.e., programmers appreciate that defects could be more easily detected with CbC). We consider three research questions to evaluate the methodologies (RQ1–2) and the tools (RQ3) qualitatively.

**RQ1:** What errors do participants make with CbC or PhV?

**RQ2:** What is the process of participants to create programs with CbC or PhV?

**RQ3:** Do participants prefer CorC or KeY?

**Participants.** Our participants were students of a software quality course at TU Braunschweig, Germany. We decided for these students because they were taught the fundamentals of software verification, and they got an introduction to both tools. They have experience in verifying methods with both tools although the specific algorithms of this experiment were new to them. We had ten participants which were divided into two groups randomly. The programming experience that was measured with an initial questionnaire [18] was 2.189 for group A and 1.791 for group B.<sup>3</sup> The experience of individuals ranged between 1.609 and 2.777. With a Mann-Whitney test, we calculated no significant difference between both groups (p-value = 0.1514). Most of the students have several years of programming experience in industry, and therefore, can be compared to junior developers. Six participants had three to seven years experience as programmer in industry, two were new programmers in larger projects, and only two never programmed in larger projects.

The participants voluntarily attended in the experiment. They knew that they took part in an experiment and that this experiment did not affect the grade of the course. Every participant was paid € 10 to create an incentive for them. Participants who solved one or both exercises also had the chance to win € 50 (i.e., one of them was randomly selected). This lottery should increase the motivation to solve the exercises by creating a realistic pressure to succeed.

**Material.** In our experiment, the participants had to implement and verify two algorithms. For every participant, we prepared a computer with an Eclipse installation that supports CorC and KeY, and contained a workspace with the two exercises. We also provided a cheat sheet containing the syntax of KeY and CorC to help the participants. In order for us to properly analyze the experiment, participants took the programming experience questionnaire before the exercises

<sup>2</sup> <https://github.com/Runge93/UserstudyCbCPhV>.

<sup>3</sup> The calculation is explained in the work by Feigenspan et al. [18]. They derived with stepwise regression testing that the experience in comparison to classmates with factor 0.441 summed up with the logical programming experience with factor 0.286 is the best indicator for programming experience.

and a user experience questionnaire afterwards. The user experience questionnaire is a combination of open questions (OQ 1–4) and the User Experience Questionnaire<sup>4</sup> (UEQ).

**OQ1:** What was better in CorC/KeY?

**OQ2:** How did you proceed with the task in CorC/KeY?

**OQ3:** Which tool would you use for verification, and why?

**OQ4:** Which tool better supports avoiding or fixing defects, and why?

UEQ is an established questionnaire which measures six properties of a product (e.g., attractiveness) by asking the user to rate the product with 26 items. Each item describes the product positively and negatively, and the user must evaluate which and to what extent one of the descriptions fits. Additionally, the workspaces were saved to analyze the created code and specifications.

**Tasks.** We used the Latin square design to arrange the participants. Group A used CorC for a `maximum element` algorithm, and KeY for `modulo`. Group B did the exercises in the same order, but each one with the other tool. We switched the order of the tools to address learning and ordering effects. We believe that an order between tools is worse than an order between exercises because we want to get insights in the usability of the tools. Additionally, the order between exercises was not varied because a split into four groups was not manageable. For each exercise, we provided a pre-/postcondition, and a task description in which we explained the purpose of the algorithm, so that the participants understood what the implementation should achieve.

The algorithm `maximum element` finds the index of the maximum element in an array. The array is assumed to be non-empty to simplify the algorithm, so that an index of the array should always be returned. The algorithm `modulo` gets two integers  $a$  and  $b$  as input and computes the two values factor and remainder for the equation  $factor * b + remainder = a$ . For the construction of the algorithm, the division and modulo operations are prohibited. Both algorithms are similar in size and cyclomatic complexity.

The tasks were designed such that a small, manageable subset of Java is sufficient to implement the algorithms. `Assignments`, `If-Then-Else`, and `While` were the only necessary statements. We excluded method calls because they complicate the verification for these two algorithms unnecessarily.

**Variables.** In our experiment, the tool is an independent variable, with the two treatments CorC and KeY. To check the correctness of the code in KeY, we reran the proof for the solution of every participant. In CorC, we checked that all nodes in the refinement hierarchy are proven. If a solution was not proven, we checked whether the code is correct with KeY and, if necessary, adjusted the specification, such as a loop invariant, to close the proof. If the code was also incorrect, we checked how many defects were in the code by adjusting the code. To evaluate the programming and verification process, we analyzed the intermediate snapshots. Here, the changes and defects were also counted in

<sup>4</sup> <https://www.ueq-online.org/>.



**Table 1.** Defects in code and specification of the final programs of participants

#Defects	KeY		CorC	
	Code	Specification	Code	Specification
Verified	2		3	
No defects	8	2	4	3
Minor defects	1	4	3	2
Major defects	1	3	1	2
Incomplete	0	1	2	3

terms of changed lines. For example, if an incorrect assignment was fixed by a participant, we count one change in the program and reduce the number of defects by one. The time needed for every exercise was measured manually. If a participant solved a task, the time was noted. After 30 min, we interrupted the participants when they were not finished.

**Deviations.** The participants assigned themselves randomly to a group by selecting one computer. We missed that the participants per groups were unequal. Group A had six participants, and group B had only four. This unequal distribution changed which exercise was done with which tool. Since we used the Latin square design, the influence should not be significant because we still had ten results for each treatment.

## 4 Results and Discussion

In this section, we present the results of our evaluation. We analyzed the data of the created programs and the answers of the questionnaire. The comparably small number of participants reduces the generalizability of the results, but allows us to evaluate the process of the participants in detail by analyzing all 347 intermediate code snapshots. This gives us anecdotal evidence to qualitatively discuss advantages and disadvantages of CbC and PhV.

### 4.1 Defects in Implementation

To answer the first research question, RQ1, what errors do participants make with CbC or PhV, we analyze defects in the program and the specification.

There are ten implementations with each tool. The defects in the code are shown in Table 1 in column two and four, numbered left-to-right from one. With KeY, eight programs were correct and two of them were verified. In one case, only a loop guard was slightly incorrect (e.g., two variables were compared with less than, but less than or equal was correct). Only one program contained major defects. We classified a program to have major defects, if we could not correct

**Table 2.** Initial and final defects in the programs of participants

Row	Initial defects	Final defects	KeY	CorC
1	0	0	6	1
2	1	0	1	1
3	2	0	1	0
4	3	0	0	1
5	4	0	0	1
6	1	1	1	0
7	2	1	0	2
8	3	1	0	1
9	>5	>5	1	1
10	Incomplete		0	2

the program with at most five changes. With CorC, four programs were correct and three of them could be verified. In three programs, a minor defect occurred, one program had numerous mistakes, but also two programs were incomplete.

In the case of intermediate specifications which needed to be provided, for both tools the results were worse. In Table 1, the defects in intermediate and loop invariant specifications are shown in column three and five. Only in two cases for KeY and three cases for CorC no defects occurred. In KeY, four specifications contained minor defects, such as a missing boundary for a control variable or an incorrect comparison of two variables. Three programs had major defects in the specification. For example, it was not properly specified which elements of the array were already examined in the `maximum element` algorithm. One participant did not create an invariant. In the case of CorC, two minor and two major defects occurred, but also three algorithms had incomplete specifications. Two of these three incomplete specifications could be explained as incomplete programs. In the third case, the algorithm was created but not specified.

To analyze the defects in more detail, we counted the defects during the programming task. In Table 2, the defects in the initial (i.e., programs at the first verification attempt) and final programs are shown. One difference between programming in KeY and CorC is that the participants in KeY started the first verification after the program was completely constructed. In CorC, some users started earlier, with incomplete programs because they could verify Hoare triples for parts of the programs that were already completely concretized. With KeY, six participants created a program without any defects (Row 1). In two cases (rows 2 and 3), one or two defects were found. One participant started with one defect, but could not find the defect (Row 6). The participant also had three defects in an intermediate result, but never found the incorrect loop guard condition. One program had more than five defects in the beginning and the end (Row 9). With CorC, only one program had no defects in the beginning (Row 1). Three participants started with one to four defects and fixed the defects (rows 2, 4, and 5). One participant who started with two defects and ended with one (Row 7), had a correct intermediate result, but inserted one defect in the final version. One participant had a result which could not be fixed easily (Row 9).

Two programs were incomplete in CorC (Row 10). Their developers started with the first refinements, but could not finalize the program in the CorC editor.

The construction of algorithms with KeY was mostly the same. The participants created a correct or nearly correct algorithm. Afterwards, a loop invariant was constructed and the program was verified. Astonishingly, no participant could verify the program on the first try even though the program was correct because the loop invariants were incorrect or too weak (e.g., for `modulo` the special case that the input parameters could be equal was not handled). The approach of the participants to get the program to a verifiable state was different. Some participants mostly changed the invariant and verified the program again. Others changed the loop and the invariant. A correct program was changed up to ten times to another correct solution, but no sufficient invariant for KeY to verify the program was found. Some participants also changed whether the loop variable was increased or decreased several times.

With CorC, the most common approach was to create the program with all refinements and specify the intermediate conditions or loop invariants in parallel. Often the program was completely refined before the first verifier call. If the verification was not possible, missing parts such as the initialization of control variables were added, assignment or conditions were changed. In three cases, the initial defects were found, but in one case, a correct intermediate program was changed to an incorrect program. The participant with the incorrect result started with a program where he forgot to decrease the control variable in the loop. Afterwards, the participant decreased the variable correctly, but the loop invariant was wrong, so the statements could not be verified. So, the program was changed again to decrease the control variable at another place in the program. In the process, the participant introduced an incorrect execution path where the variable is not decreased. Two other participants started with a loop, but forgot the initialization of necessary variables. This mistake was recognized during the exercise.

In summary, both tools in some cases lead to correct and verified programs. Small defects occurred with both tools, but in CorC, we observed incomplete programs. If the program could not be verified, participants mostly changed the loop guards, the loop body, or loop invariants. The changes in the code are fewer with CorC than with KeY. If a program could not be verified, the problem was in most cases an insufficient loop invariant or a wrong loop guard. With PhV, most participants created correct code in the first place. As shown in Table 2, only three defects were found in the process in total. With CbC, the users started mostly with a defective program and found twelve defects in total. This higher number of found defects may be explained with better tool support in CorC, but also with the higher number of existing defects. With PhV, only four defects existed by excluding the completely wrong program. Thus 75% of the defects were found. For CbC, there are 15 defects in total, so 80% have been found.

**RQ1.** Comparing the defects in code, participants made similar errors with both techniques (e.g., incorrect loop guard), but they made fewer and mostly minor errors with PhV. This could be explained with the familiar environment of standard Java with JML. The two incomplete programs in CorC can be explained by problems interacting with the tool. Thus in total, more correct programs

were created with PhV than with CbC. That more programs were verified with CbC anyway is interesting. One explanation could be that programs with CbC were less changed. The participants might have thought more about the program instead of changing the program by trial-and-error. Due to the similar correction rates of defects for both tools, we cannot confirm a negative influence of CbC in the programming process, but we should further investigate why more defect programs with CbC exist.

## 4.2 Analysis of Programming Procedure

From the intermediate snapshots, we can evaluate the programming procedure by analyzing the changes and defects in code and intermediate specification, and missing program or specification parts. We analyzed 20 solutions containing between 9 and 39 snapshots. We excluded the incomplete and entirely incorrect cases because we could not count wrong or missing parts with the same scale as for the other cases. In the following, three typical results are shown.

In Fig. 1, we show the graph of a participant solving the `maximum element` task in CorC. The participant started the verification process with two missing lines of code and two missing intermediate specification lines. The participant also had two defects in the intermediate specification. Overall, 25 steps were taken by the participant to achieve the correct solution. In the first 13 steps, the program and the specification were changed, but no defects were fixed. In Step 14, the invariant of the program was corrected. The special case that there can be more than one maximum element in the array was included in the invariant. The next steps were used to verify the program, until the participant realized that the initialization of variables was missing. After this fix in Step 21, the program was verified.

In Fig. 2, the process to construct the `maximum element` algorithm in KeY is shown. The participant started with a correct program where the invariant was missing. After introducing the invariant with a defect, the participant changed the code and the invariant during the whole task without finding a sufficient invariant. The program was changed to iterate the array from forward to backward and vice versa several times. The main reason that the program could not be verified was that the invariant did not specify which elements of the array were already visited. There were similar cases with KeY where also only the invariant was wrong. The code and intermediate specifications were changed by most participants during their development process. There were two participants who mostly changed the invariant instead of the code.

In Fig. 3, we show a graph of a user developing the `modulo` algorithm in CorC. The participant started with one defect in the code, an incorrect loop guard, and two missing specification parts, the invariant and an intermediate condition. In the first steps, the participant tried to verify the whole program without changing it. Then, the missing specifications were added, but both were wrong. In the invariant, the comparison operator was the wrong way around, and the intermediate condition was too weak (i.e., it was not specified that the correct factor was found). The specification was changed until step twelve, then the participant tried to verify the program again. As this did not lead to a

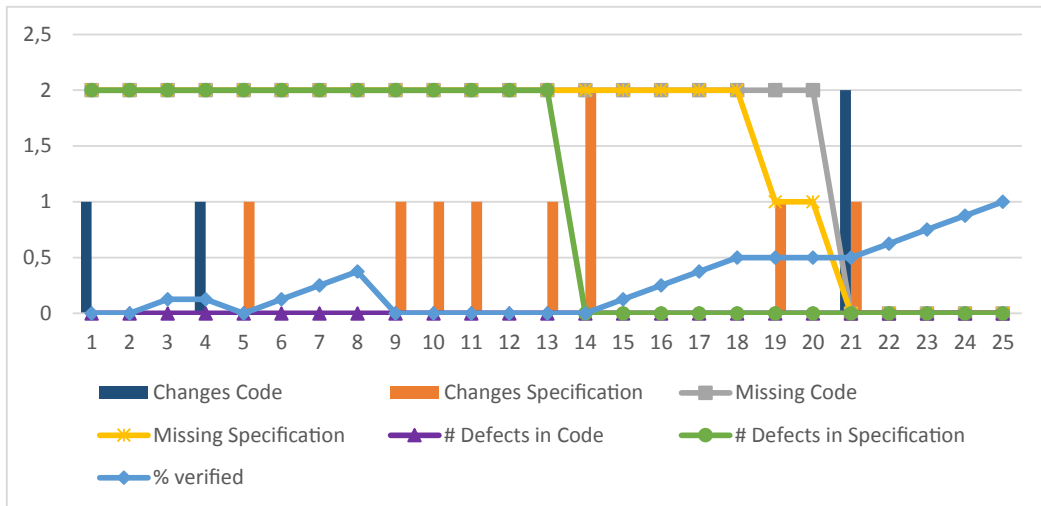


Fig. 1. Process to construct maximum element algorithm in CorC

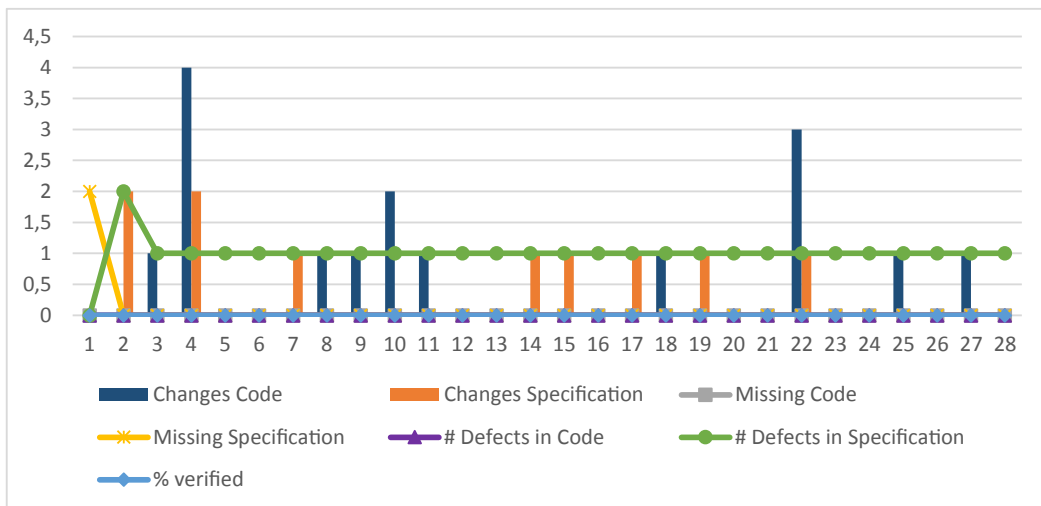


Fig. 2. Process to construct maximum element algorithm in KeY

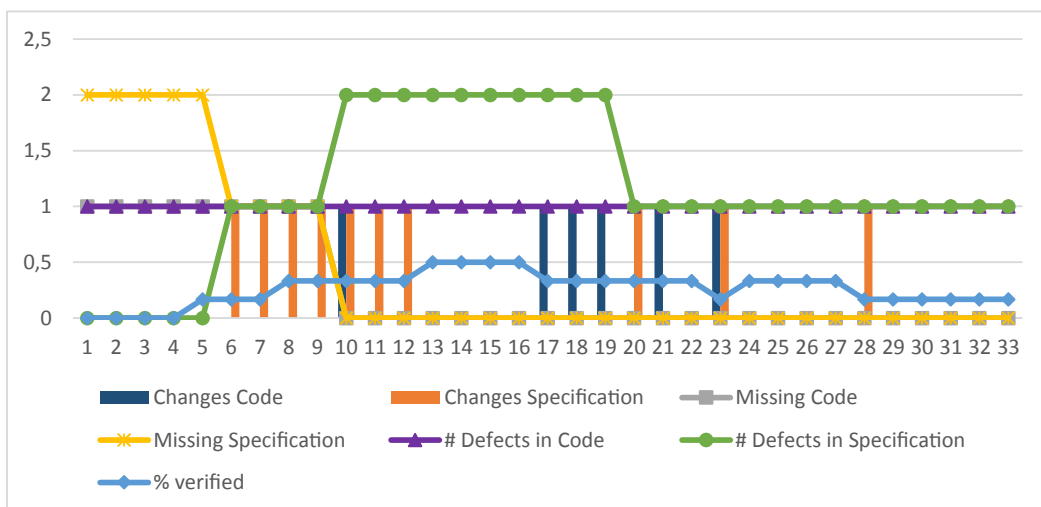
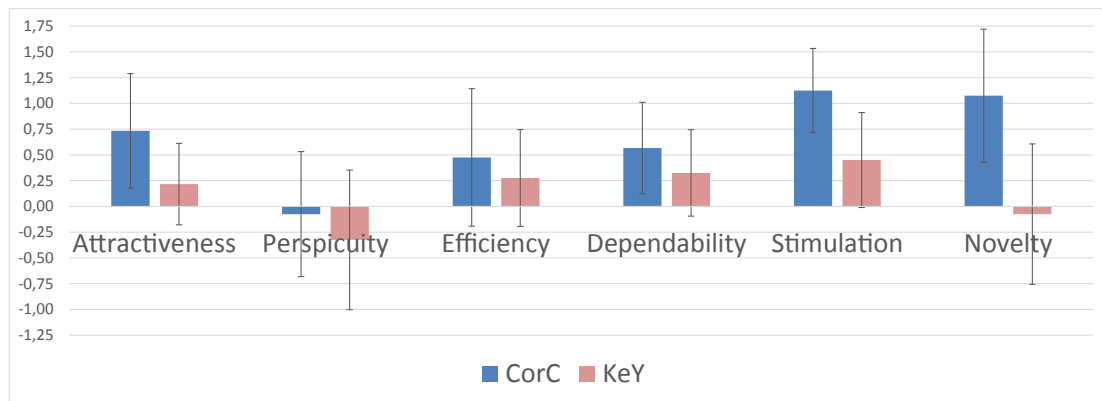


Fig. 3. Process to construct modulo algorithm in CorC



**Fig. 4.** Results of the user experience questionnaire

solution, the code and specification were changed again. The wrong comparison in the invariant was found, but the other two problems remained until the end.

**RQ2.** With the detailed analysis of all 347 program snapshots, we can discuss the programming process of the participants. We saw that correct programs were changed several times if they could not be verified, and surprisingly remained correct. The participants did not realize that only the intermediate specification was insufficient. They need better tool support to pinpoint the defects in code or specification. We also noticed a non-monotonic construction process for both techniques. By monotonic, we mean that a program is specified, constructed, and then verified to be correct. An example for the non-monotonic construction process with CbC and PhV are the trial-and-error changes in specification and code. For example with PhV, the users changed to iterate the array from forward to backward in several cases. With CbC, the users verified a correct part of the program, but changed it if they could not verify the complete program. In comparison to PhV, participants using the CbC approach changed the code less. Furthermore, a correct specification may favor the finding of mistakes in the program. Often defects were found after a correct loop invariant was introduced. In our evaluation, all programs with correct specifications had no defects.

### 4.3 User Experience

The results of the UEQ are presented in Fig. 4. The answers of the participants were evaluated according to the six measurements: attractiveness, perspicuity, efficiency, dependability, stimulation, and novelty. The scale is between +3 and -3 for each item. Overall, the average answers of the participants are higher for CorC. For perspicuity both tools got a negative mean value. KeY also has a negative result for novelty. We measured a significant difference with the T-test.<sup>5</sup> Stimulation ( $p = 0.0457$ ) and novelty ( $p = 0.0274$ ) are significantly different.

<sup>5</sup> Statistical hypothesis test to compare two independent samples which are normally distributed.



For the open questions, we clustered the answers to analyze whether the participants had similar experiences. The results are shown in the following.

**OQ1.** *What was better in CorC/KeY?* Five to six participants valued the clarity of CorC. They also valued the good feedback of CorC to spot the defects in the program because the program is split into individually provable statements. On the negative side, the unfamiliar syntax and the handling of the tool were mentioned. In the case of KeY, the well-known Java and JML syntax was appreciated by nearly all participants. Two participants also valued the clarity of KeY. One participant disliked the bad error messages of KeY. Another one mentioned that KeY gives more information about the problem, but this follows from the design of the experiment. CorC uses KeY as back-end for verification, but we suppressed the KeY editor on purpose in CorC because the verification problems for the implemented algorithms should be small enough to be verified automatically [34]. In the normal configuration, CorC can deliver the same information by opening the proof in KeY. In summary, the known syntax in KeY was an advantage, but the participants appreciated the better potential in CorC to find the defects because the program was decomposed into provable statements.

**OQ2.** *How did you proceed with the task in CorC/KeY?* In KeY, all participants created the code first, then they created the loop invariant and verified the program. One participant emphasized that the program was inferred from the postcondition. In CorC, the common case was to construct the code stepwise. Two participants explicitly mentioned that they created the program in CorC first, then specified the program. Two others started with the specification in CorC. In contrast to KeY, the participants wrote specifications only in CorC before or during the construction of the code.

**OQ3.** *Which tool (CorC/KeY) would you use for verification, and why?* Five participants decided to use CorC for verification. They appreciated the clarity. Two participants mentioned the support to verify and debug individual statements. One participant highlighted the reflective coding process that is encouraged by CorC. Four participants decided to use KeY. They liked the familiar environment and syntax. As in the first question, one participant mentioned that KeY gives more information. There is no clear trend towards one tool.

**OQ4.** *Which tool (CorC/KeY) better supports avoiding or fixing defects, and why?* Most participants decided for CorC to avoid or fix defects. They appreciated that defects are assigned to individual statements, therefore, it was easier to understand the problem. One participant mentioned that the stepwise construction helped to create correct programs. For both tools, some participants indicated that defects were detected and only correct code could be verified. Although nearly the same number of participants would use KeY or CorC for verification, most participants wanted to use CorC to find or fix defects in the coding process. That defects were associated to specific statements was well received by the participants.

**RQ3.** The third research question, whether participants prefer CorC or KeY, can be answered with the results of the questionnaire. The participants preferred



KeY because of the familiar syntax, and CorC for the better feedback if there were defects in the code. This leads to a balanced vote on which tool the participants would use for verification. Interestingly, the participants voted in favor of CorC when it comes to finding and fixing defects. This should be further investigated; what keeps participants from using CorC even though they mention that it helped better to find defects. With the answers of the participants and the analysis of the snapshots, we can also confirm how the participants worked on the tasks. In KeY, the program was developed, and afterwards the specification was constructed. So, the code was mostly correct in the first place. In CorC, they had different approaches. They interleaved coding and specification or started with the specification. This results in starting the verification earlier with incomplete or incorrect programs. Surprisingly, nobody complained about the additional specification effort in CorC.

#### 4.4 Threats to Validity

In our experiment, we had only 10 participants. This reduces the generalizability of the results, but allowed us to analyze all 347 versions of program snapshots in detail. The participants were all students of a software quality course. We could ensure that all students had the required theoretical and practical precognition. They are no experts in verifying software, but smaller tasks, such as those of our experiment, were solved before by the participants in class. Most students also have part-time jobs in companies, so the results are generalizable to junior developers. The motivation of the students is doubtful, but the lottery gave an incentive to accomplish the tasks. Another limitation for the experiment was the limited time. Most participants have accomplished to write correct code, but only five out of twenty algorithms were also verified. With more time it is possible that more algorithms would have been verified. We only used two small size exercises in our experiment, and therefore, cannot generalize the results to bigger problems. The results of the experiment also depend on our introduction of the tools—though we tried to introduce both tools equally without bias to the students.

## 5 Related Work

In the literature, tool support for verification was previously evaluated, but PhV was not compared with CbC.

Spec# is an extension of the programming language C# to annotate code with pre-/postconditions and verify the code using the theorem prover Boogie [10, 11]. Barnett et al. [11] explained their lessons learned of constructing this verification framework. In contrast, we focus on how users solve programming and specification tasks. Petiot et al. [33] examined how programmers could be supported when a proof is not closed. They implemented tool support that categorizes the failure and gives counter examples to improve the user feedback. This idea is complementary to the CbC method by pinpointing the failure to

a small Hoare triple, which was appreciated by the participants in this study. Johnson et al. [23] interviewed developers about the use of static analysis tools. They came to the same result as we did that good error reporting is crucial for developers. Hentschel et al. [21] studied the influence of formal methods to improve code reviews. They detected a positive impact of using the symbolic execution debugger (SED) to locate errors in already existing programs. This setup is different to our evaluation where the participants had to program actively. The KeY tool [12, 13] was already evaluated to get insight into how participants use the tool interactively. In contrast, we wanted to evaluate the automatic part of KeY because we think that most users do not have a theoretical background to verify a program interactively.

Besides CorC and KeY, there are other programming languages and tools using specification for program verification. For example Eiffel [28, 29] with the verifier AutoProof [24, 35], SPARK [9], Whiley [32], OpenJML [15], Frama-C [16], VCC [14], Dafny [26, 27], VeriFast [22], and VerCors [5]. These languages and verification tools can be used to compare CbC with post-hoc verification. As we only used a subset of the Java language in our experiment (comparable to a simple while language), the difference to other programming languages is minimal, and we expect similar results for those tools as with KeY.

A related CbC approach is the Event-B framework [1]. Here, automata-based systems are specified, and can be refined to concrete implementations. The Rodin platform [3] implements the Event-B method. For the predecessor of Event-B, namely the B method, Atelier B [2] is used to prove correctness. The main difference to CorC is the different abstraction level. CorC uses source code with specification rather than automata-based systems. The CbC approaches of Back [8] and Morgan [30] are related to CbC by Dijkstra, and it would be interesting to evaluate these approaches in comparison to our CbC tool in a future study. For example, ArcAngel [31] could be used as an implementation of Morgan's refinement calculus. Back et al. [6, 7] build the invariant based programming tool SOCOS. They start explicitly with the specification of not only pre-/postconditions but also invariants before the coding process. In their experiment, they discovered that good tool support is needed and that invariants are found iteratively by refining an incomplete and partly wrong invariant; an insight which we can confirm.

## 6 Conclusion and Future Work

We compared correctness-by-construction and post-hoc verification by using the tools CorC and KeY. Participants could create and verify programs, but the majority failed to create invariants that were strong enough. When a program could not be verified, trial-and-error was the most popular strategy to fix the program. Regarding user experience, KeY and CorC were both considered useful to verify software, but the good feedback of CorC was explicitly highlighted. Nevertheless, the defects in the programs with CorC indicate that the participants need more time to get used to CorC.

We evaluated the user study qualitatively to get insights in how users create verified programs. For future work, we could repeat the experiment with more participants to get quantitative data about defects in the programs. Furthermore, our insights about the trial-and-error programming process could be used to improve the usability of both tools.

**Acknowledgment.** We would like to thank Alexander Knüppel and Domenik Eichhorn for their help with the user study. The hints and suggestions of Alexander helped to construct the final version of the study. Thanks to Domenik for setting up the tools.

## References

1. Abrial, J.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-B. *Int. J. Softw. Tools Technol. Transf.* **12**(6), 447–466 (2010)
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification-The KeY Book: From Theory to Practice, vol. 10001. Springer, Heidelberg (2016). <https://doi.org/10.1007/978-3-319-49812-6>
5. Amighi, A., Blom, S., Darabi, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Verification of concurrent systems with VerCors. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 172–216. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-07317-0\\_5](https://doi.org/10.1007/978-3-319-07317-0_5)
6. Back, R.-J.: Invariant based programming: basic approach and teaching experiences. *Formal Aspects Comput.* **21**(3), 227–244 (2009)
7. Back, R.-J., Eriksson, J., Myreen, M.: Testing and verifying invariant based programs in the SOCOS environment. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 61–78. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-73770-4\\_4](https://doi.org/10.1007/978-3-540-73770-4_4)
8. Back, R.-J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (2012)
9. Barnes, J.G.P.: High Integrity Software: The Spark Approach to Safety and Security. Pearson Education, London (2003)
10. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. *Commun. ACM* **54**(6), 81–91 (2011)
11. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-30569-9\\_3](https://doi.org/10.1007/978-3-540-30569-9_3)
12. Beckert, B., Grebing, S., Böhl, F.: A usability evaluation of interactive theorem provers using focus groups. In: Canal, C., Idani, A. (eds.) SEFM 2014. LNCS, vol. 8938, pp. 3–19. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-15201-1\\_1](https://doi.org/10.1007/978-3-319-15201-1_1)

13. Beckert, B., Grebing, S., Böhl, F.: How to put usability into focus: using focus groups to evaluate the usability of interactive theorem provers. *Electron. Proc. Theor. Comput. Sci.* **167**, 4–13 (2014)
14. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03359-9\\_2](https://doi.org/10.1007/978-3-642-03359-9_2)
15. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20398-5\\_35](https://doi.org/10.1007/978-3-642-20398-5_35)
16. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Framac-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33826-7\\_16](https://doi.org/10.1007/978-3-642-33826-7_16)
17. Dijkstra, E.W.: *A Discipline of Programming*. Prentice Hall, Upper Saddle River (1976)
18. Feigenspan, J., Kästner, C., Liebig, J., Apel, S., Hanenberg, S.: Measuring programming experience. In: 2012 IEEE 20th International Conference on Program Comprehension (ICPC), pp. 73–82. IEEE (2012)
19. Gries, D.: *The Science of Programming*. Springer, Heidelberg (1987)
20. Hall, A., Chapman, R.: Correctness by construction: developing a commercial secure system. *IEEE Softw.* **19**(1), 18–25 (2002)
21. Hentschel, M., Hähnle, R., Bubel, R.: Can formal methods improve the efficiency of code reviews? In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 3–19. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-33693-0\\_1](https://doi.org/10.1007/978-3-319-33693-0_1)
22. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-17164-2\\_21](https://doi.org/10.1007/978-3-642-17164-2_21)
23. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: *Proceedings of the 2013 International Conference on Software Engineering*, pp. 672–681. IEEE Press (2013)
24. Khazeev, M., Rivera, V., Mazzara, M., Johard, L.: Initial steps towards assessing the usability of a verification tool. In: Ciancarini, P., Litvinov, S., Messina, A., Sillitti, A., Succi, G. (eds.) SEDA 2016. AISC, vol. 717, pp. 31–40. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-70578-1\\_4](https://doi.org/10.1007/978-3-319-70578-1_4)
25. Kourie, D.G., Watson, B.W.: *The Correctness-by-Construction Approach to Programming*. Springer, Heidelberg (2012). <https://doi.org/10.1007/978-3-642-27919-5>
26. Leino, K.R.M.: Specification and verification of object-oriented software. *Eng. Methods Tools Softw. Saf. Secur.* **22**, 231–266 (2009)
27. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-17511-4\\_20](https://doi.org/10.1007/978-3-642-17511-4_20)
28. Meyer, B.: Eiffel\*: a language and environment for software engineering. *J. Syst. Softw.* **8**(3), 199–246 (1988)
29. Meyer, B.: Applying “design by contract”. *Computer* **25**(10), 40–51 (1992)
30. Morgan, C.: *Programming from Specifications*, 2nd edn. Prentice Hall, Upper Saddle River (1994)
31. Oliveira, M.V.M., Cavalcanti, A., Woodcock, J.: ArcAngel: a tactic language for refinement. *Formal Aspects Comput.* **15**(1), 28–47 (2003)

32. Pearce, D.J., Groves, L.: *Whiley: a platform for research in software verification*. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) *SLE 2013*. LNCS, vol. 8225, pp. 238–248. Springer, Cham (2013). [https://doi.org/10.1007/978-3-319-02654-1\\_13](https://doi.org/10.1007/978-3-319-02654-1_13)
33. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: *Your proof fails? Testing helps to find the reason*. In: Aichernig, B.K.K., Furia, C.A.A. (eds.) *TAP 2016*. LNCS, vol. 9762, pp. 130–150. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-41135-4\\_8](https://doi.org/10.1007/978-3-319-41135-4_8)
34. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: *Tool support for correctness-by-construction*. In: Hähnle, R., van der Aalst, W. (eds.) *FASE 2019*. LNCS, vol. 11424, pp. 25–42. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2)
35. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: *AutoProof: auto-active functional verification of object-oriented programs*. In: Baier, C., Tinelli, C. (eds.) *TACAS 2015*. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46681-0\\_53](https://doi.org/10.1007/978-3-662-46681-0_53)
36. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.: *Correctness-by-construction and post-hoc verification: a marriage of convenience?* In: Margaria, T., Steffen, B. (eds.) *ISoLA 2016*. LNCS, vol. 9952, pp. 730–748. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-47166-2\\_52](https://doi.org/10.1007/978-3-319-47166-2_52)

### **A.3. Teaching Correctness-by-Construction and Post-hoc Verification — The Online Experience**



# Teaching Correctness-by-Construction and Post-hoc Verification – The Online Experience

Tobias Runge<sup>1(✉)</sup>, Tabea Bordis<sup>1(✉)</sup>, Thomas Thüm<sup>2(✉)</sup>, and Ina Schaefer<sup>1(✉)</sup>

<sup>1</sup> TU Braunschweig, Brunswick, Germany  
{tobias.runge,t.bordis,i.schaefer}@tu-bs.de

<sup>2</sup> University of Ulm, Ulm, Germany  
thomas.thuem@uni-ulm.de

**Abstract.** Correctness of software is an important concern in many safety-critical areas like aviation and the automotive industry. In order to have skilled developers, teaching formal methods is crucial. In our software quality course, we teach students two techniques for correct software development, post-hoc verification and correctness-by-construction. Due to Covid, the last course was held online. We present our lessons learned of adapting the course to an online format on the basis of two user studies; one user study held in person in 2019 and one online user study held after the online course. For good online teaching, we suggest the use of accessible (web-)tools for active participation of the students to compensate the advantages of teaching in person.

## 1 Introduction

Development of correct software is a concern which is becoming increasingly important. In areas like aviation and the automotive industry, where human lives depend on it, software safety requirements are considerably higher. Therefore, formal methods should be taught to young software developers, so that they learn a reasonable approach to develop correct programs, instead of hacking programs into correctness. With this skill of correct software development, developers can avert major errors in software projects by specifying and verifying the safety-critical parts [35]. Additionally, a development process that includes verification can reduce overall development time because most software is correct from the start, which reduces maintenance time and effort. Besides the prevalent post-hoc verification (PhV) approach, where software is verified after implementation, correctness-by-construction (CbC) [23] is an approach where software is incrementally refined from a specification. In CbC, each refinement step guarantees the correctness of the whole programs. CbC expands the repertoire of programmers with a formal reasoning style that prevents errors in the first place.

Teaching formal methods, the correct specification and verification of programs, is the topic of the Master course Software Quality 2 at TU Braunschweig,



Germany. In the course, students learn the basics of deductive software verification and correctness-by-construction on the practical example of specifying and verifying Java code. In the corresponding exercises, the students solve tasks using corresponding tools. In contrast to previous years, we offered the course of last term online (due to the pandemic).

A difficulty in teaching formal methods is that courses are based on a lot of formal background which may discourage students. Therefore, we highlight the benefit to integrate practical experiences of practical tool usage in teaching that help students to consolidate the taught topics [17]. With tools, students receive immediately feedback if their found solutions are correct. They can also work on larger tasks that are not doable on pen-and-paper. A problem here is the effort to install various tools on different machines, especially when students use their own machines due to online teaching. Tools should be easy to install or web-based such that students enjoy active participation in lectures and exercises of formal method courses. Some good examples for tool support are KeY [3], Whiley [30], and Dafny [25].

Besides an experience report on our online course, we evaluate the learning success of the students with two user studies. We compare the results of an online user study with an earlier user study conducted in the same course, but in person [33]. In the qualitative user study, we evaluate how students solved tasks with two verification techniques post-hoc verification and correctness-by-construction. They used the tools KeY [3] (as instance of a PhV tool) and (Web)CorC [32] (as instance of a CbC tool). Therefore, our user study has four quadrants. We compare CbC with PhV and online with in person courses. With the data from these two studies, we share our lessons learned in transferring the course to an online format, we discuss the quality of the online course, and point out challenges and opportunities for improving online courses in the future. We also compare the web version WebCorC with the previous version CorC to determine important aspects of good tool support. Furthermore, we compare how well students interact with CbC and PhV by collecting their user feedback.

As a result, we confirmed the findings of the first user study. The students made fewer defects in the code with PhV than with CbC, but overall the results are worse than in the first user study. This indicates a worse learning outcome due to the online format. The qualitative questionnaire was answered in favor of CbC. The students liked the structured reasoning of CbC and rated the support provided by the CorC tool as more suitable for finding defects than KeY for PhV. For online teaching, we detect that easily accessible tool support is beneficial for participation in exercises. Additionally, courses on formal methods should be interactive to encourage student participation, thus we discuss how to improve online teaching.

## 2 Related Work

Teaching formal methods has been discussed by many researchers [11, 14, 17, 26]. They discuss their teaching experiences and evaluate the learning success

of students with respect to different tools and teaching strategies. In detail, Liu et al. [26] highlight the benefits of a good mix of pen-and-paper and tool supported exercises. On paper, students consolidate what they learned without being supported by a tool, and with tools they increase their productivity and learn to analyze defects in the specifications or programs. The interest of students also rises if they can solve exercises on tools and get positive feedback by verifying programs. Creuse et al. [14] mention that teaching by example is beneficial for an easier and more practical entry into formal methods. That the students demand immediate and good feedback on their specification or verification process and want to understand clearly occurring problems is identified by Catano [11]. We differ from this related work [11, 14, 17, 26] that does not examine the aspects of online teaching. In this paper, we discuss new challenges regarding online teaching by comparing our course with a previous course held in person.

With respect to the user study, we compare it with related work that evaluated the usage of verification tools. Petiot et al. [31] evaluated the interaction of programmers with verification tools. The authors analyzed how programmers can be supported when they encounter an open proof goal. To improve user feedback, they categorize defects and calculate counter examples. In the work of Johnson et al. [22], developers were interviewed about their usage of static analysis tools. They also recognized that developers need good error reporting. The influence of formal methods in code reviews was studied by Hentschel et al. [20]. In their study, the symbolic execution debugger (SED) had a positive impact on the location of defect in programs. KeY was also evaluated to analyze how participants interact with the tool during the verification process [9, 10]. Back [5] evaluated in an experiment that good tool support is necessary to develop correct software with a refinement based approach. Additionally, he discovered an iterative procedure to refine an incomplete or partially incorrect invariant to a final solution, an insight that we confirm in our first user study [33]. In our user study, we focus more on the usability of the tools during program constructions, how programmers utilize the tools and adapt their programming procedure. In our study, we used KeY as automatic verifier. Thus, we excluded the expertise on interactive proving from our study. We focused on the development of correct programs guided by a specification. In our study, the participants have to implement the programs by themselves.

### 3 Teaching Formal Methods – Software Quality 2

In this section, we describe the structure of the formal methods course Software Quality 2 at TU Braunschweig. We compare the previous course held in person with the current course in an online format. The goal of this course is to teach students deductive post-hoc verification and correctness-by-construction such that students are able to construct correct programs with these approaches. The course is named Software Quality 2, as we also offer a course Software Quality 1 that focuses on testing software.

*In Person Course.* The course in person is divided into two parts, 12 lectures with 11 corresponding exercises, each one lasting 90 min. The students attending this course are mostly Master students that had at least two courses in programming, and three courses in theoretical computer science. Normally, between 20 and 50 students attend the course. The lectures are a presentation on topics like design by contract, software model checking, sequent calculus, deductive verification, specifying programs with JML, verifying programs with KeY, and constructing correct programs with CbC. The presentations include some intermediate questions to the audience (e.g., to complete examples) and questions in the end to consolidate the lessons learned. We provide a video of each lecture, so students can prepare the exams by rewatching specific lessons. In the videos, the slides and the audio of the lecture are recorded.

The exercises are divided between pen-and-paper exercises for writing first-order logic and using the sequent calculus, and tool-supported exercises. For the tool-supported exercises, we use different tools: OpenJML [13] to show testing with JML annotated code, Java Pathfinder [19] for software model checking, KeY [3] for program verification, and CorC [32] for correct-by-construction software development. All these tools are pre-installed on machines at the university such that exercises are performed smoothly. The exercises are mostly interactive such that the students solve the tasks and present the solutions, and these solutions are discussed with the audience. This structure of the exercises should consolidate knowledge better than a frontal presentation of solutions.

The course exam is oral. We have a small group of students such that oral exams are feasible. In these exams, we can check whether students understood the topics of this course and can answer cross-cutting questions, and whether they can apply learned techniques to solve code specification and verification tasks. Oral exams are more time-consuming than written exams, but as a teacher one gets better feedback on whether students have understood the content.

*Online Course.* The setting for the online course is the same as for the previous courses in person: 12 lectures with 11 exercises with weekly meetings covering the same topics. The number of students slightly increased to around 60 students. We upload videos of the recorded lectures of the previous year. Additionally, we give a short recap of the topic followed by a discussion where students can ask questions in the weekly meeting. For the exercises, we upload exercise sheets with tasks that have to be prepared as homework. If the task includes the use of tools, we add an instruction for the installation and usage. In the weekly online session for the exercises, the students are asked to present their solutions which are discussed with the audience afterwards. Thereby, we use Google Docs documents that can be edited by everyone in the session to collect and store the correct answers for exam preparation.

To keep the interactive character of the exercise in the online course, we use the same tools in the exercises as we do for the course in person (i.e. OpenJML, Java Pathfinder, KeY, and CorC). Due to the format of an online session, we could not monitor whether students were actually actively participating. For the tools, we tried to find the easiest way with as few steps as possible for

the installation. However, with most of the tools we had some problems with the installation due to outdated documentations or the need of specific JDK or Eclipse versions such that finding a good solution was time-consuming.

The oral exams are held online in the video conference system provided by our university. The student has to attend with camera such that we can check that the right person is taking the exam and that no other persons are in their room. An advantage of taking the oral exam online is that we are able to include practical tasks using tools introduced in the exercises. To omit difficulties in the installation, we installed the tools on our computer and shared the screen. The students then have to explain what they would do and what result they expect.

## 4 Verification Techniques and Tool Support

We compare in our user study, how students solve tasks using post-hoc verification (PhV) and correctness-by-construction (CbC). We briefly introduce PhV and CbC in the following.

### 4.1 Post-hoc Verification

The post-hoc verification process [3], verifies the correctness of programs after the implementation. A prover checks that the implementation complies with the pre-/postcondition specification. PhV does not give a development guideline, such that programmers can freely implement the programs as long as the specification is in the end fulfilled. This free process decreases the time to construct a first (potentially faulty) version of a program, but can increase the time to construct a verified version, as it is more likely that defects occur in the code [35].

As an instance, KeY [3] verifies the correctness of Java programs that are specified with the Java modelling language (JML). Starting from a specified program, KeY symbolically executes the programs and closes the remaining proof obligations (semi-)automatically. As we are focusing on the programming and specification aspects in our user study, we use KeY as an automatic tool. This goes along with most programmers not having a theoretical background to verify programs interactively.

Besides KeY, there are a number of tools in the area of specification and program verification: the language Eiffel [27] with the verifier AutoProof [34], the languages SPARK [8], Dafny [25], and Whiley [30], and the tools OpenJML [13], Frama-C [15], VCC [12], VeriFast [21], and VerCors [4]. All languages and tools are candidates to be compared with the CbC methodology, but we decided for KeY because of the previous familiarity of our study participants. Since we used only a subset of the Java language without method calls or custom objects, the difference to other programming languages is minimal.

### 4.2 Correctness-by-Construction

*Correctness-by-construction* [16,23,28] is a methodology to incrementally construct correct programs. Starting with a pre-/postcondition specification and an

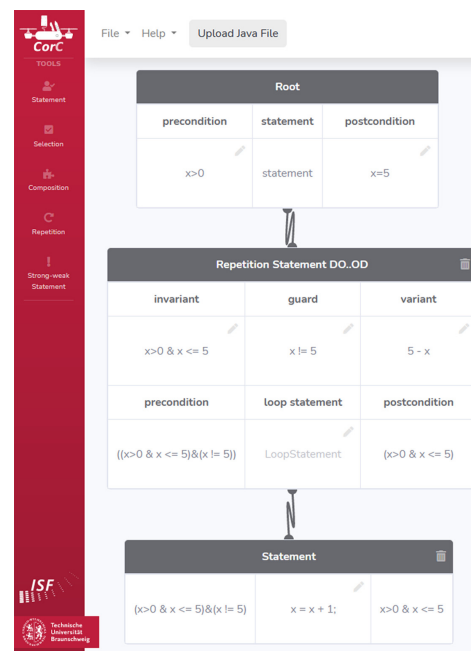
initially abstract program, refinement rules are applied to create an implementation that fulfills the specification. The correctness is guaranteed by the rules if specific side conditions for their applicability hold. Dijkstra [16] and Kourie and Watson [23] identified that the CbC process guides programmers to a correct implementation that has low defects rates and is of better structure than a program ad hoc hacked to correctness. A disadvantage of CbC is the fine-grained refinement process that programmers must adhere to. This complicates program construction for inexperienced programmers, but the fine-grained development with the explicit specification in each node raises awareness for defects in the mind of the programmer [35].

Besides the CbC approach proposed by Kourie and Watson [23], there are other refinement based approaches that guarantee the correctness of the program under development. In the Event-B framework [1], specified automata-based systems are refined to concrete implementations. It is implemented in the Rodin platform [2]. In comparison to the CbC approach used here, the abstraction level is different. CbC uses specified source code instead of automata as main artifact. Morgan [28] and Back [7] proposed also related CbC approaches. Morgan's refinement calculus is implemented in the tool ArcAngel [29]. Back et al. [5,6] developed the tool SOCOS. In comparison to CbC, SOCOS starts with invariants additionally to a pre-/postcondition specification.

The tool CorC [32] implements the CbC process in a graphical and textual editor. CorC supports developers by offering refinement rules as proposed by Kourie and Watson [23] to develop programs and checking the correctness of each applied refinement with a program verifier KeY [3]. In CorC, a programmer builds stepwise a correct method by getting feedback directly when one refinement is not correct, for example, when the programmer specifies an invariant that is not satisfied at the beginning of the loop.

WebCorC<sup>1</sup> is an adaption of CorC [32]. Similar to CorC, we decided for the graphical editor in WebCorC because of the student feedback collected during the Software Quality courses. The graphical editor helps students learn the CbC approach by visualizing all important aspects of specifying and refining a program into a correct result. CorC

is implemented in Java in the Eclipse framework. In WebCorC, we transferred the graphical editor using a client-server structure, reusing the logic of CorC on server side, but redeveloping the graphical editor as web-frontend. In comparison to CorC, the implementation of WebCorC has no detailed feedback in a console



**Fig. 1.** Program construction in WebCorC

<sup>1</sup> <https://www.isf.cs.tu-bs.de/WebCorC/>.



when a refinement cannot be proven. This feedback was added only after the user study.

In Fig. 1, we show a program construction in WebCorC. At the top, we specify in the first gray node the program under development. The precondition states that an integer  $x$  is greater than zero. In the postcondition,  $x$  should be equal to 5. We solve this problem by introducing a loop statement in the first refinement step, called repetition in WebCorC. We introduce a repetition statement for illustration purposes. Of course, an assignment directly solves the problem. For the repetition, we need additional specification: a loop guard, a loop invariant, and a variant. We continue the loop as long as  $x$  is not equal to 5. The loop body introduced in the next refinement step, the third node at the bottom, increments  $x$  by one. Both refinement steps are checked by WebCorC to be correct.

## 5 User Study Design

In this section, we describe the design of our user study that was conducted online after the end of the Software Quality 2 course. The goal of this evaluation is to compare the results of the students with the results of a previous study that was held in person. Therefore, we adopt the design of the previous study [33]. We want to get insights into the learning success of the students whether the online course and the use of WebCorC leads to noticeable differences in the outcome. To better compare both studies, we explain the commonalities and differences in the user study design in the following. Note that we used CorC in the first user study and WebCorC in the second user study. If we talk about both tools, we write (Web)CorC.

### 5.1 General User Study Design

The user study is designed such that students solve two programming tasks each with a different tool. We compare correctness-by-construction and post-hoc verification with the tools (Web)CorC and KeY. Starting with a pre-/postcondition specification, an algorithm should be implemented and verified.

**Objective.** We want to evaluate whether the CbC approach has a positive or a negative impact on the programming results. We consider the following two research questions to evaluate the verification approaches and tools.

**RQ1:** What kind of errors do participants make with CbC and PhV?

**RQ2:** To which degree do participants prefer CbC over PhV?

To evaluate the usability of CbC and PhV, we take the user experience questionnaire (UEQ), and ask the questions OQ1–OQ8:

**OQ1:** How do you rate your overall work with WebCorC from 1 (very bad) to 5 (very good)?

**OQ2:** What is your general process when solving tasks with WebCorC/KeY?

- OQ3:** Do you prefer a web-frontend over the Eclipse environment and why?  
**OQ4:** Were there any specific obstacles during the task execution process?  
**OQ5:** Is the construction of a program by modeling through a refinement structure helpful and why?  
**OQ6:** Do you prefer WebCorC or KeY in general and why?  
**OQ7:** Which of these two tools would you use for verification and why?  
**OQ8:** Which tool better supports avoiding or fixing defects and why?

The UEQ [24] is a standardized test to measure six usability properties of a tool. A participant is asked to rate the tool with 26 items. Each item is a pair of adjectives that describe the tool, one negative and one positive adjective. The user can rate on a 7-point Likert-scale which of the adjectives and to what extent fits more. The range for the answers is between  $+3/-3$ .

**General Design Decisions.** The user study is limited to 90 min. To compare both tools, each participant should implement an algorithm with each tool. We set 30 min per task. Thus, the algorithms should be implementable and verifiable in this time frame. We decide for algorithms with a size of under ten lines, but including a loop to have participants writing loop invariants. We give the pre-/postcondition specification of the algorithms so that all participants have the same starting point. This reduces the divergence and lead to comparable programming results. Writing the pre-/postcondition would also cost too much time in this experiment. We decide for the tools (Web)CorC and KeY because the participants have experience with these tools which increases the expressiveness of this study. The material of the user study is published on GitHub.<sup>2</sup>

**Tasks.** Two algorithms must be correctly implemented and verified. The algorithm `minimum element` calculates the index of the minimum element in an array. The array is non-empty to omit the special case from the algorithm. The algorithm `modulo` calculates the remainder from two input integers; a dividend  $a$  and a divisor  $b$ . In the algorithm, division and modulo operators are prohibited. The algorithms are similar in size and cyclomatic complexity.

We design the tasks to be small enough to be doable in the time frame. We also design them such that both (Web)CorC and KeY can be used to implement them correctly. For both tasks, assignments, conditional statements and loops where sufficient.

The groups of participants are arranged with the Latin square design. Group A uses (Web)CorC for the first task, and PhV afterwards. Group B does the same tasks but the tools in different order. The tools are switched to address the possibility of learning effects by forcing a specific tool order.

**Participants.** The participants are students at TU Braunschweig, Germany attending the Software Quality course. These students were taught the fundamentals of software verification, and they learned to use the tools (Web)CorC and KeY. During the course they implement, specify, and verify methods with both tools. We analyzed the programming experience of the participants with a questionnaire [18]. To weaken the restraint against a group of students, we had

<sup>2</sup> <https://github.com/Runge93/UserstudyCbCPhV>.



several students with two to five years of programming experience in industry. Therefore, the participants can be compared with junior developers. The students freely attended the user study. We told them the goal of this experiment, and we offered a monetary payment for attendance. We raffled two times € 25.

**Variables.** We have the tools as independent variable in our user study, with the treatments CbC and PhV. The correctness of the task were checked with KeY using the automatic mode. For CbC, we checked the task by reverifying each refinement step. When some task was not verifiable, we manually checked for defects in the program or specification. These defects were counted by line.

## 5.2 Differences in the First and Second User Study

In the first user study, we have 10 participants in two groups who have no significant difference in the programming experience [33]. In the second user studies, we have 13 participants. With a programming experience questionnaire [18], we measure a similar experience in both groups. A value of 2.137 for group A and 2.550 for group B<sup>3</sup>. With a Mann-Whitney test, no significant difference between the two groups is measured.

In the first user study, we prepare machines at the university with CorC and KeY. They directly implement both tasks in the Eclipse IDE. Here, they can interact with KeY directly to get feedback about the verification status. In the second user study, we prepare a workspace where the participants can develop one of the algorithms with WebCorC. For the PhV process, they can use their preferred IDE. When they want to verify the algorithm, they upload it and get feedback about the success of the verification. This process can be repeated until a verified result is achieved. As the participants in the second study only upload files in the post-hoc verification tasks and do not interact with KeY directly, we abandon the UEQ for the tool KeY.

In the first user study, we monitored all participants in a controlled experiment in person. In the second user study, we adapt this by having an audio conference. Due to legal restrictions, we cannot use proctoring tools.

## 6 Results and Discussion

In this section, we show the results of our user study. We evaluate the implementations of each participant by looking at the final result. We focus on a qualitative evaluation of the programming procedure and results of CbC and PhV. Additionally, we evaluate the answers of our questionnaire (UEQ and OQ1–OQ8) to complete the discussion.

### 6.1 Defects in Implementation and Specification

To answer the first research question, we analyze defects in the code and the specification. By code, we refer to the implemented algorithm without the specification. By specification, we refer to auxiliary annotations such as loop invariants.

<sup>3</sup> The calculation is explained in the work by Feigenspan et al. [18].

**Table 1.** Defects in code and specification of the final programs of participants

#Defects	PhV 1 <sup>st</sup>		CbC 1 <sup>st</sup>		PhV 2 <sup>nd</sup>		CbC 2 <sup>nd</sup>	
	Code	Spec.	Code	Spec.	Code	Spec.	Code	Spec.
No defects	8	2	4	3	9	1	2	1
Minor defects	1	7	3	4	4	10	4	5
Major defects	1	0	1	0	0	0	0	0
Incomplete	0	1	2	3	0	2	7	7

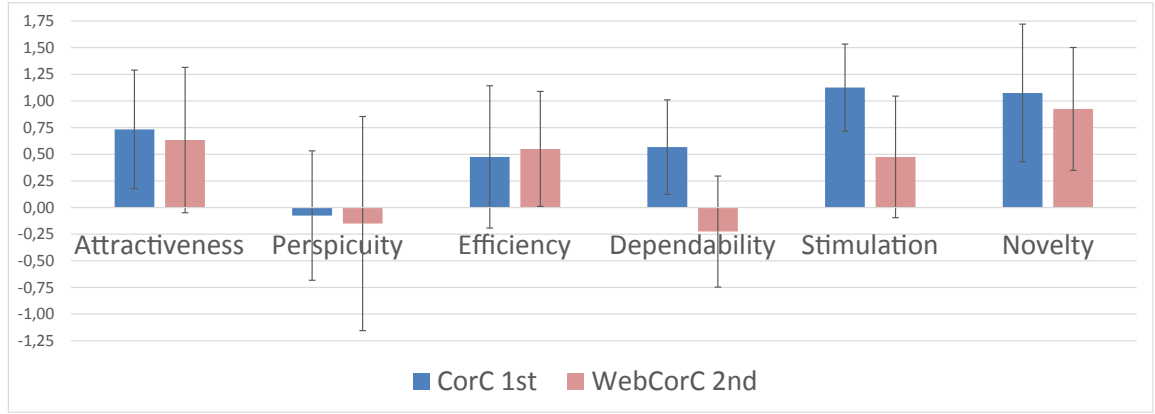
We classify a program to have major defects, if the program cannot be corrected without rewriting the algorithm. Otherwise, we classify it to have minor defects. The same classification applies for defects in the specification.

Table 1 shows the defects the participants have in their final result when they finished a task in the first and second user study. When we compare CbC and PhV, the participants have fewer coding defects with PhV than with CbC in both user studies. The coding results for PhV and CbC are generally better in the first user study, we have more results without defects for both approaches. Across all participants, a typical defect is a loop guard with a wrong logical comparison operator. With CbC, a recurring problem is that participants forget to initialize variables correctly. In both studies participants have incomplete results in the program for CbC. In the second study, seven participants have not completed the program for CbC.

When we compare defects in the auxiliary specification, more participants have no defects with CbC in the first user study compared with the results for PhV. In the second user study, for each approach only one participant has no defects. In general, the specification results are better in the first user study. More participants have no defects with PhV and CbC. Typical defects with PhV are a missing variant or missing checks whether variables stay in a specific boundary (e.g., out of bounds checks in arrays). With CbC, a common specification defect is that the invariant does not hold initially or after the last loop iteration. However, the participants do not forget the variant when they specify a loop. In both user studies and with both approaches, we have incomplete specifications. Five incomplete results of the auxiliary specification are due to incomplete code.

## 6.2 User Experience

For the evaluation of the user experience, we show the results of the user experience questionnaire in Fig. 2. The blue results for CorC are from the first user study [33], the red results for WebCorC are from the second study. The answers of the participants are combined into six measurements: *attractiveness*, *perspicuity*, *efficiency*, *dependability*, *stimulation*, and *novelty*. Except for *efficiency*, the results are better in the first user study. The largest differences are in the scales of *stimulation* and *dependability*. The *stimulation* is rated lower because some participants rate WebCorC *demotivating*. Participants also rate WebCorC as



**Fig. 2.** Results of the user experience questionnaire

*unpredictable* which results in a negative score for *dependability*. The items *easy to learn/difficult to learn* and *complicated/easy* are answered differently resulting in a big variance for the *perspicuity* measurement in the second user study.

For question OQ1-OQ8, common answers of the participants are summarized in Table 2. Some participants dislike the limited feedback of WebCorC in comparison to CorC, but they prefer the web-frontend due to the easy accessibility. The general process of solving tasks is split between writing specification or the program first. When comparing WebCorC with KeY, the majority of participants prefer WebCorC to solve verification tasks because of the structured process. The participants in favor of KeY argue that they are more familiar with textual programming.

### 6.3 Discussion of the Research Questions

**RQ1.** When we compare the defects in code, the participants have similar defects in both approaches (e.g., incorrect loop guards or incomplete invariants), but they have fewer defects with PhV. A possible reason is the familiar environment of writing Java code in a textual editor. Overall, we have worse results in the second study. Regarding the complete results, we explain the difference between both studies with the better feedback of CorC in comparison to WebCorC such that students can find defects more easily. Another reason is that we monitored active participation in the exercises in person. For the online course, we cannot confirm this. It seems that the students were better prepared in the first user study. We noticed that considerably more participants in the second user study have not the necessary knowledge to construct programs with CbC. Some students may not have participated in the exercises and may not have familiarized themselves with (Web)CorC.

**RQ2.** We answer the second research question, whether participants prefer CbC or PhV. Participants like the familiar programming style with PhV, but the majority prefer (Web)CorC over KeY. The participants mention that CorC has better and fine-grained console feedback which helps detecting defects during program construction. In the previous study, the participants highlight the good

**Table 2.** Answers for the questions OQ1–OQ8

Question	Answer
OQ1	On average, the participants rate the work with WebCorC slightly worse (2.1/5)
OQ2	They think about the solution first. The group of participants is split between first writing code or specification
OQ3	CorC has more functionality. Web-frontend is easier to access and system independent
OQ4	Some participants are not experienced enough to interact with WebCorC
OQ5	Participants find defects in corner cases with WebCorC. They divide the problem into smaller blocks. CbC rules are too restrictive. Some are unfamiliar with graphical programming
OQ6	Six answers in favor of (Web)CorC. CorC has better feedback than WebCorC. Two participants prefer KeY because of the familiar programming style
OQ7	Six answers in favor of (Web)CorC. Two answers in favor of KeY
OQ8	Six answers in favor of (Web)CorC, mostly because of the better feedback for verification results. Two answers in favor of KeY, as KeY shows the whole proof tree

feedback for each refinement step, which is not implemented in WebCorC yet. With better feedback, they would prefer WebCorC over CorC due to the easier handling and installation. Surprisingly, nobody complains about the additional specification effort in CbC.

Compared to the first UEQ shown in Fig. 2, we get slightly worse results in the second study. The main reason is worse user feedback for WebCorC in comparison to CorC. This insight coincides with the answers of the open questions in both user studies. Thus, participants rate WebCorC more demotivating, unpredictable, and harder to learn because CorC is more advanced. Due to the online course, it was also harder to teach the tools to the students. Students asked fewer questions, therefore, problems were not discovered that also arose during the user study (e.g. the correct initialization of variables). In person, problems stand out more quickly and can be easily explained. Nevertheless, the participants in both studies prefer (Web)CorC over KeY. Considering that the students have more defects with (Web)CorC, the students seem to factor in that they like the CbC approach for correct software development. The main reason against (Web)CorC is that participants are more familiar with the programming style in KeY. A limitation that is likely due to the shorter time of working with the CbC process.

#### 6.4 Threats to Validity

*External Validity.* The user studies had 10 and 13 participants. With this limited number of participants, the generalizability of the results is restricted, but we

were able to analyze the programming results of each participant in detail. The participants are all Computer Science students that learned verification in the Software Quality 2 course. Therefore, they are not experts in verification, but should be able to solve smaller examples as the ones asked in this study. Through their statements in the programming experience questionnaire, most students can be compared to junior developers. Furthermore, the small algorithms reduce the generalizability for larger algorithmic problems. Regarding the time frame of a course, a longer study was not feasible.

*Internal Validity.* The motivation of the participants and their effort of solving the tasks could not be monitored due to the online version of this user study. As the time was limited for each task, most solutions were not verified completely. With additional time, it would be possible for more algorithms to be verified.

## 7 Lessons Learned for Online Teaching

In this section, we conclude the paper by summarizing our lessons learned for online teaching. The first three findings are based on the results in the user studies. The last three also include our experiences from the online course.

**Procedure of Software Development.** By analyzing the questionnaire and the programming results of the user study, we notice that students are mostly hacking programs into correctness. By teaching the correctness-by-construction approach, we enable students to think of the specification and the corner cases first, before starting to program. This is well-received by the students, but the approach needs time to be adopted.

**Accessibility of Tools.** In the questionnaire, students highlight that tools should be easy to install for online teaching. If a tool needs many installation steps or has a high potential to fail on some machines, students will not actively participate in exercises. Students that are not able to solve the CbC tasks indicate missing knowledge in (Web)CorC. Also, tools should be freely available such that students are not excluded because they cannot afford the tool. Many tools that we use during our lectures are Eclipse plug-ins. For Eclipse plug-ins, the easiest way to install them is by using an update site. However, for some tools, the update sites are not accessible anymore or only work with specific versions of Eclipse or JDK. That has to be checked before a course.

**Feedback of Tools.** From the questionnaire, we know that tools should give detailed and fine-grained feedback if errors occur in the development process. Without feedback, the finding of defects during new tasks gets frustrating such that students tend to give up faster online. This confirms results in the literature [11, 26].

Besides of the user studies, we also collect feedback during the courses. Good teaching is characterized by active participation of students [17, 26]. As students are more quickly distracted online, we describe how to improve the online course such that students actively participate. Regarding the fact that we have more

programming defects in the second user study, we still have to improve the online course to be as good as the course in person.

**Breakout Rooms.** During the online exercises, we found that including tasks where students can work in small groups in breakout rooms increases the number of actively participating students. This holds, especially when the teacher is not constantly in the same room and the students can work together on a task, which has not been prepared in advance. Generally, breakout rooms also help students to connect with each other and build learning groups for exams which became more difficult during the pandemic.

**Interactions in an Online Setting.** In online courses, it is way more important that students are willing to follow the lecture and to participate in exercises. In the results of the user study, we encounter several students that indicate missing background knowledge for the tasks. To prevent this, we derive the following best practices for online teaching: Students should attend exercises with cameras which increases attention. Students should be integrated into lectures by asking questions. When videos and slides are provided in addition to a lecture, students can consolidate what has been learned. Exercises with voluntary tasks are working only for a minority of students. Other students will attend the exercises unprepared. So exercises need to be mandatory or could provide bonus points for the final exam.

**Openness to Novel Approaches.** Students are open minded for new techniques and tools as we can see from our experiences with (Web)CorC. As teachers, we have to ensure that new topics are introduced interactively and with examples. However, when the new technique is not introduced properly, students will not consider it for future tasks and fall back to old familiar approaches. We want to ensure that formal methods are not taught for the sake of the course, but be anchored in the mind of young computer scientists. So the introduction of the new techniques needs to be thorough, well illustrated using meaningful examples, and supported by accessible tools.

**Acknowledgments.** We thank Huu Cuong Nguyen and Malena Horstmann for their help in preparing and conducting the user study.

## References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. *STTT* **12**(6), 447–466 (2010)
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification-The KeY Book: From Theory to Practice, vol. 10001. Springer, Heidelberg (2016). <https://doi.org/10.1007/978-3-319-49812-6>



4. Amighi, A., Blom, S., Darabi, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Verification of concurrent systems with VerCors. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 172–216. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-07317-0\\_5](https://doi.org/10.1007/978-3-319-07317-0_5)
5. Back, R.J.: Invariant based programming: basic approach and teaching experiences. FAOC **21**(3), 227–244 (2009)
6. Back, R.-J., Eriksson, J., Myreen, M.: Testing and verifying invariant based programs in the SOCOS environment. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 61–78. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-73770-4\\_4](https://doi.org/10.1007/978-3-540-73770-4_4)
7. Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (2012)
8. Barnes, J.G.P.: High Integrity Software: The Spark Approach to Safety and Security. Pearson Education (2003)
9. Beckert, B., Grebing, S., Böhl, F.: A usability evaluation of interactive theorem provers using focus groups. In: Canal, C., Idani, A. (eds.) SEFM 2014. LNCS, vol. 8938, pp. 3–19. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-15201-1\\_1](https://doi.org/10.1007/978-3-319-15201-1_1)
10. Beckert, B., Grebing, S., Böhl, F.: How to put usability into focus: using focus groups to evaluate the usability of interactive theorem provers. EPTCS **167**, 4–13 (2014)
11. Cataño, N.: Teaching formal methods: lessons learnt from using Event-B. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 212–227. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-32441-4\\_14](https://doi.org/10.1007/978-3-030-32441-4_14)
12. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03359-9\\_2](https://doi.org/10.1007/978-3-642-03359-9_2)
13. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20398-5\\_35](https://doi.org/10.1007/978-3-642-20398-5_35)
14. Creuse, L., Dross, C., Garion, C., Hugues, J., Huguet, J.: Teaching deductive verification through FRAMA-C and SPARK for non computer scientists. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 23–36. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-32441-4\\_2](https://doi.org/10.1007/978-3-030-32441-4_2)
15. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Framac. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33826-7\\_16](https://doi.org/10.1007/978-3-642-33826-7_16)
16. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Hoboken (1976)
17. Divasón, J., Romero, A.: Using Krakatoa for teaching formal verification of Java programs. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 37–51. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-32441-4\\_3](https://doi.org/10.1007/978-3-030-32441-4_3)
18. Feigenspan, J., Kästner, C., Liebig, J., Apel, S., Hanenberg, S.: Measuring programming experience. In: ICPC, pp. 73–82. IEEE (2012)
19. Havelund, K., Pressburger, T.: Model checking Java programs using Java pathfinder. STTT **2**(4), 366–381 (2000)



20. Hentschel, M., Hähnle, R., Bubel, R.: Can formal methods improve the efficiency of code reviews? In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 3–19. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-33693-0\\_1](https://doi.org/10.1007/978-3-319-33693-0_1)
21. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-17164-2\\_21](https://doi.org/10.1007/978-3-642-17164-2_21)
22. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: ICSE, pp. 672–681. IEEE Press (2013)
23. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer, Heidelberg (2012)
24. Laugwitz, B., Held, T., Schrepp, M.: Construction and evaluation of a user experience questionnaire. In: Holzinger, A. (ed.) USAB 2008. LNCS, vol. 5298, pp. 63–76. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-89350-9\\_6](https://doi.org/10.1007/978-3-540-89350-9_6)
25. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-17511-4\\_20](https://doi.org/10.1007/978-3-642-17511-4_20)
26. Liu, S., Takahashi, K., Hayashi, T., Nakayama, T.: Teaching formal methods in the context of software engineering. ACM SIGCSE Bull. **41**(2), 17–23 (2009)
27. Meyer, B.: Eiffel: a language and environment for software engineering. JSS **8**(3), 199–246 (1988)
28. Morgan, C.: Programming from Specifications, 2nd edn. Prentice Hall, Hoboken (1994)
29. Oliveira, M.V.M., Cavalcanti, A., Woodcock, J.: ArcAngel: a tactic language for refinement. FAOC **15**(1), 28–47 (2003)
30. Pearce, D.J., Groves, L.: Whiley: a platform for research in software verification. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 238–248. Springer, Cham (2013). [https://doi.org/10.1007/978-3-319-02654-1\\_13](https://doi.org/10.1007/978-3-319-02654-1_13)
31. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: Your proof fails? Testing helps to find the reason. In: Aichernig, B.K.K., Furia, C.A.A. (eds.) TAP 2016. LNCS, vol. 9762, pp. 130–150. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-41135-4\\_8](https://doi.org/10.1007/978-3-319-41135-4_8)
32. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool support for correctness-by-construction. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 25–42. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2)
33. Runge, T., Thüm, T., Cleophas, L., Schaefer, I., Watson, B.W., et al.: Comparing correctness-by-construction with post-hoc verification—a qualitative user study. In: Sekerinski, E. (ed.) FM 2019. LNCS, vol. 12233, pp. 388–405. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-54997-8\\_25](https://doi.org/10.1007/978-3-030-54997-8_25)
34. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46681-0\\_53](https://doi.org/10.1007/978-3-662-46681-0_53)
35. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.: Correctness-by-construction and post-hoc verification: a marriage of convenience? In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 730–748. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-47166-2\\_52](https://doi.org/10.1007/978-3-319-47166-2_52)

## **A.4. Traits: Correctness-by-Construction for Free**



# Traits: Correctness-by-Construction for Free

Tobias Runge<sup>1,2(✉)</sup>, Alex Potanin<sup>3(✉)</sup>, Thomas Thüm<sup>4(✉)</sup>,  
and Ina Schaefer<sup>1,2(✉)</sup>

<sup>1</sup> TU Braunschweig, Braunschweig, Germany

<sup>2</sup> Karlsruhe Institute of Technology, Karlsruhe, Germany  
{tobias.runge, ina.schaefer}@kit.edu

<sup>3</sup> Australian National University, Canberra, Australia  
alex.potanin@anu.edu.au

<sup>4</sup> University of Ulm, Ulm, Germany  
thomas.thuem@uni-ulm.de

**Abstract.** We demonstrate that traits are a natural way to support correctness-by-construction (CbC) in an existing programming language in the presence of traditional post-hoc verification (PhV). With Correctness-by-Construction, programs are constructed incrementally along with a specification that is inherently guaranteed to be satisfied. CbC is complex to use without specialized tool support, since it needs a set of refinement rules of fixed granularity which are additional rules on top of the programming language.

In this work, we propose TraitCbC, an incremental program construction procedure that implements correctness-by-construction on the basis of PhV by using traits. TraitCbC enables program construction by trait composition instead of refinement rules. It provides a programming guideline, which similar to CbC should lead to well-structured programs, and allows flexible reuse of verified program building blocks. We introduce TraitCbC formally and prove the soundness of our verification strategy. Additionally, we implement TraitCbC as a proof of concept.

## 1 Introduction

*Correctness-by-Construction* (CbC) [19, 22, 30, 37] is a methodology that incrementally constructs correct programs guided by a pre-/postcondition specification.<sup>1</sup> CbC uses small tractable refinement rules where in each refinement step, an abstract statement (i.e., a hole in the program) is refined to a more concrete implementation that can still contain some nested abstract statements. While

---

<sup>1</sup> The approach should not be confused with other CbC approaches such as CbyC of Hall and Chapman [24]. CbyC is a software development process that uses formal modeling techniques and analysis for various stages of development (architectural design, detailed design, code) to detect and eliminate defects as early as possible [13]. We also exclude data refinement from abstract data types to concrete ones during code generation as for example in Isabelle/HOL [23].

refining the program, the correctness of the whole program is guaranteed through the check of conditions in the refinement rules. The construction ends when no abstract statement is left. Through the structured reasoning discipline that is enforced by the refinement rules, it is claimed that program quality increases and verification effort is reduced [30,50].

Despite these benefits, CbC has a drawback: the refinement rules extend the programming language (i.e., refinements are an additional linguistic construct to transform programs). Special tool support [42] is necessary to introduce the CbC refinement process to a programming language. Additionally, the predefined rules have a fine granularity such that for every new statement the programmer adds to the program, an application of a refinement rule is necessary. Consequently, the concepts of CbC (e.g., abstract statements and refinement rules) increase the effort and necessary knowledge of the developer to construct programs.

*Post-hoc verification* (PhV) is another approach to develop correct programs. A method is verified against its pre- and postconditions after implementation. In practice, it often happens that a program is constructed first, with the objective of verifying it later [50]. This can lead to tedious verification work if the program is not well-structured. An example is the difficult search for the many reasons preventing the verification of a method to be completed: an incorrect specification, an incorrect method, or inadequate tool support. Therefore, a structured programming approach is desirable to construct programs which are amenable to software verification.

In this work, we use *traits* [20] to overcome the drawbacks of CbC (complex programming style using external refinement rules) and introduce a programming guideline for an incremental trait-based program construction approach that guarantees that the resulting trait-based program is correct-by-construction. TraitCbC is based on PhV. With TraitCbC, the same programs can be verified as with PhV, but in addition, TraitCbC introduces an explicit program construction approach. It utilizes the flexibility of traits, which is beneficial for scenarios as incremental development [18] and the development of software product lines [10,15].

*Traits* [20] are a flexible object-oriented language construct supporting a rich form of modular code reuse orthogonal to inheritance. A trait contains a set of *concrete* or *abstract* methods (i.e., the method has either a body or has no body), independent of any class or inheritance hierarchy.<sup>2</sup> Traits are independent modules that can be composed into larger traits or classes. When traits are composed, the resulting code contains all methods of all composed traits. To verify traits, Damiani et al. [18] proposed a modular and incremental *post-hoc verification* process. Each method in every trait is verified in isolation by showing that the method satisfies its *contract* [35]. Then, during the composition of traits, it has to be checked whether a method implemented in one trait is compatible with the abstract method with the same signature in another trait. That means,

---

<sup>2</sup> The term trait has been used by many programming languages: Java interfaces with default methods are a good approximation for what has been called trait in the literature, while Scala traits are mixins [21], and Rust traits are type classes [46].

a concrete method has to satisfy the specification of the abstract method. A concrete method with a weaker precondition and a stronger postcondition fulfills the contract of the abstract method (cf. Liskov substitution principle [34]).

A developer using TraitCbC starts by implementing a method (e.g., a method **a**) in a first trait. Similar to CbC, the method can contain holes that are refined in subsequent steps. A hole in TraitCbC is an abstract method (e.g., an abstract method **b**) that is called in method **a**; that is, a call to an abstract method corresponds to an abstract statement in CbC. In the next step, one of these new abstract methods (e.g., **b**) is implemented in a second trait, again more abstract methods can be declared for the implementation. Similar to PhV, it must be proven that the implemented methods satisfy their specifications. Afterwards, the traits are composed; the composition operation checks that the contract of the concrete method **b** in the second trait fulfills the contract of the abstract method **b** in the first trait. This incremental process stops when the last abstract method is implemented, and all traits are composed.

The main result of our work is the discovery that traits intrinsically enable correctness-by-construction. This work is not about pushing verification forward in the sense of adding more expressive power. TraitCbC realizes a refinement-based program development approach using pre-/postcondition contracts and method calls instead of refinement rules and abstract statements as in CbC. Refinement rules in the form of trait composition exist as a direct concept of the programming language instead of being a program transformation concept. Additionally, each method implemented in the refinement process can be reused by composing traits in different contexts (i.e., already proven methods can be called by new methods under construction). This is advantageous compared to the limited reuse potential of methods in class-based inheritance. Finally, TraitCbC is parametric w.r.t. the specification logic. Thus, a language with traits can adopt the proposed CbC methodology.

## 2 Motivating Example

In this section, we go through an example of how our development process enables CbC using traits.

*Incremental Construction of MaxElement.* We use a sample object-oriented language in the code examples. We construct a method `maxElement` that finds the maximum element in a list of numbers. A list has a head and a tail. Only non-empty lists have a maximum element. This is explicit in the precondition of our specification, where we require that the list has at least one element. In the postcondition, we specify that the result is in the list and larger than or equal to every other element. A method `contains` checks that the result is a member of the list. In the first step, we create a trait `MaxETrait1` that defines the abstract method `maxElement`. The method `maxElement` is abstract, i.e., equivalent to an abstract statement in CbC.

```

1  trait MaxETrait1 {
2    @Pre: list.size() > 0
3    @Post: list.contains(result) &
4          (forall Num n: list.contains(n) ==> result >= n)
5    abstract Num maxElement(List list);
6  }

```

In the second step in trait `MaxETrait2`, we implement the method `maxElement` using two abstract methods. We introduce an `if-elseif-else`-expression where the branches invoke abstract methods. The guards check whether the list has only one element or whether the current element is larger than or equal to the maximum of the rest of the list. The abstract method `accessHead` returns the current element, and the abstract method `maxTail` returns the maximum in the remaining list. So, we recursively search the list for the largest element by comparing the maximum element of the list tail with the current element until we reach the end of the list.

```

1  trait MaxETrait2 {
2    @Pre: list.size() > 0
3    @Post: list.contains(result) &
4          (forall Num n: list.contains(n) ==> result >= n)
5    Num maxElement(List list) =
6      if (list.size() == 1) {accessHead(list)}
7      elseif (accessHead(list) >= maxTail(list))
8        {accessHead(list)}
9      else {maxTail(list)}
10
11   @Pre: list.size() > 0
12   @Post: result == list.element()
13   abstract Num accessHead(List list);
14
15   @Pre: list.size() > 1
16   @Post: list.tail().contains(result) &
17         (forall Num n: list.tail().contains(n) ==> result >= n)
18   abstract Num maxTail(List list);
19 }

```

The correct implementation of the method `maxElement` can be guaranteed under the assumptions that all introduced abstract methods are correctly implemented. Similar to PhV, a program verifier conducts a proof of method `maxElement` and uses the introduced specifications of the methods `accessHead` and `maxTail`. When the proof succeeds, we know that the first method is correctly implemented. In our incremental `CbCTrait` process, we verify each method implementation directly after construction; and so we are able to reuse each implemented method in the following steps (e.g., by calling the method in the body of other methods).

We now compose the developed traits to complete the first refinement step. To perform the composition `MaxETrait1 + MaxETrait2`, we check that the specification of the method `maxElement` fulfills the specification of the abstract method in the first trait (cf. Liskov substitution principle [34]). In this case, this means checking that:

`MaxETrait1.maxElement(..).pre ==> MaxETrait2.maxElement(..).pre` as well as:  
`MaxETrait2.maxElement(..).post ==> MaxETrait1.maxElement(..).post`.

When the composition of two verified traits is successful, the result is also a verified trait. Note that the composed trait does not need to be verified directly by a program verifier in `TraitCbC` because it is correct by construction. In this example, the specifications are the same, thus checking for a successful composition is trivial, but this is

not generally the case. In particular, the logic needs to take into account ill-founded specifications and recursion in the specification. We discuss the difficulties of handling those cases in the technical report [41].

The methods `accessHead` and `maxTail` are implemented in the next two refinement steps in traits `MaxETrait3` and `MaxETrait4`<sup>3</sup>. As we implement a recursive method, the method `maxTail` calls the `maxElement` method, thus `maxElement` is introduced as an abstract method in this trait. We have to verify that the method `accessHead` satisfies its specification using a program verifier. Similarly, we have to verify the correctness of the method `maxTail`.

```

1  trait MaxETrait3 {
2    @Pre: list.size() > 0
3    @Post: result == list.element()
4    Num accessHead(List list) = list.element()
5  }

1  trait MaxETrait4 {
2    @Pre: list.size() > 1
3    @Post: list.tail().contains(result) &
4           (forall Num n: list.tail().contains(n) ==> result >= n)
5    Num maxTail(List list) = maxElement(list.tail())
6
7    @Pre: list.size() > 0
8    @Post: list.contains(result) &
9           (forall Num n: list.contains(n) ==> result >= n)
10   abstract Num maxElement(List list);
11 }
    
```

As before, all traits are composed, and it is checked that the specifications of the concrete methods fulfill the specifications of the abstract ones. As we have no contradicting specifications for the same methods, the composition is well-formed. The final program `MaxE` is as follows.

```

1  class MaxE = MaxETrait1 + MaxETrait2 + MaxETrait3 + MaxETrait4
    
```

*Advantages of TraitCbC.* As shown in the example, `TraitCbC` enables the `CbC` programming style without the need of external refinement rules. In classical `CbC`, when designing a unit of code, the programmer has to proceed with atomic steps of a pre-defined granularity. In contrast, in `TraitCbC` the programmer is free to divide a unit of code in any granularity, by including as many auxiliary methods as needed to bring the verification to an appropriate granularity. `TraitCbC` helps to construct code in fine-grained steps which are more amenable for verification than single more complex methods. If the programmer chooses to not include any auxiliary methods at all, this is essentially the same as the traditional post-hoc verification style. In the example above, we could implement the method `maxElement` in one step without the intermediate step that introduces the two abstract methods `accessHead` and `maxTail`.

Additionally, the already proven auxiliary methods in traits can be reused. For example, if we want to implement a `minElement` method, we could reuse already implemented traits to reduce the programming and verification effort. The method `minElement` is implemented in the following in trait `MinE` with one abstract method.

---

<sup>3</sup> The methods could also be implemented in one trait.



The specification of the method `accessHead` is the same as for the method `accessHead` above, so `MaxETrait3` can be reused. In this example, we show the flexible granularity of `TraitCbC` by directly implementing the else branch, instead of introducing an auxiliary method as for `maxElement`.

```

1  trait MinE {
2    @Pre: list.size() > 0
3    @Post: list.contains(result) &
4           (forall Num n: list.contains(n) ==> result <= n)
5    Num minElement(List list) =
6      if (list.size() == 1) {accessHead(list)}
7      elseif (accessHead(list) <= minElement(list.tail()))
8        {accessHead(list)}
9      else {minElement(list.tail())}
10
11   @Pre: list.size() > 0
12   @Post: result == list.element()
13   abstract Num accessHead(List list);
14 }

```

The correctness of `minElement` is verified with the specifications of the method `accessHead`. By composing `MinE` with `MaxETrait3`, we get a correct implementation of `minElement`. Note how this verification process supports abstraction: as long as the contracts are compatible, methods can be implemented in different styles by different programmers to best meet non-functional requirements while preserving the specified observable behavior [9]. A completely different implementation of `maxElement` can be used if it fulfills the specification of the abstract method `maxElement` in trait `MaxETrait1`. This decoupling of specification and corresponding satisfying implementations facilitates an incremental development process where a specified code base is extended with suitable implementations [18].

### 3 Object-Oriented Trait-Based Language

In this section, we formally introduce the syntax, type system, and flattening semantics of a minimal core calculus for `TraitCbC`. We keep this calculus for `TraitCbC` parametric in the specification logic so that it can be used with a suitable program verifier and associated logic. The presented rules to compose traits are conventional. The focus of our work is to enable a `CbC` approach using traits that programmers can easily adopt. Therefore, we present the calculus to prove soundness of `TraitCbC`, but focus on the presentation of the advantages of incremental trait-based programming in this paper. Indeed, languages with traits and with a suitable specification language intrinsically enable incremental program construction. For the sake of completeness, reduction rules of `TraitCbC` are presented in the technical report [41].

#### 3.1 Syntax

The concrete syntax of our core calculus for `TraitCbC` is shown in Fig. 1, where non-terminals ending with ‘s’ are implicitly defined as a sequence of non-terminals, i.e.,  $vs ::= v_1 \dots v_n$ . We use the metavariables  $t$  for trait names,  $C$  for class names and  $m$  for method names. A program consists of trait and class definitions. Each definition has a name and a trait expression  $E$ . The trait expression can be a *Body*, a trait name,

a composition of two trait expressions  $E$ , or a trait expression  $E$  where a method is made abstract, written as  $E[\text{makeAbstract } m]$ . A *Body* has a flag `interface` to define an interface, a set of implemented interfaces  $Cs$  and a list of methods  $Ms$ . Methods have a method header  $MH$  consisting of a specification  $S$ , the return type, a method name, and a list of parameters. Methods have an optional method body. In the method body, we have standard expressions, such as variable references, method calls, and object initializations. For simplicity, we exclude updatable state. Field declarations are emulated by method declarations, and field accesses are emulated by method calls.

The specification  $S$  in each method header is used to verify that methods are correctly implemented. The specification is written in some logic. In our examples, we will use first-order logic (cf. the example in Sect. 2). A well-formed program respects the following conditions:

Every *Name* in  $Ds$  must be unique so that  $Ds$  can be seen as a map from names to trait expressions. Trait expressions  $E$  can refer to trait names  $t$ . A well-formed  $Ds$  does not have any circular trait definitions like  $t = t$  or  $t_1 = t_2$  and  $t_2 = t_1$ . In a *Body*, all names of implemented interfaces must be unique and all method names must be unique, so that *Body* is a map from method names to method definitions. In a method header, parameters must have unique names, and no explicit parameter can be called `this`.

### 3.2 Typing Rules

In our type system, we have a typing context  $\Gamma ::= x_1 : C_1 \dots x_n : C_n$  which assigns types  $C_i$  to variables  $x_i$ . We define typing rules for our three kinds of expressions:  $x$ , method calls, and object initialization. We combine typing and verification in our type checking  $\Gamma \vdash e : C \dashv P_0 \models P_1$ . This judgment can be read as: under typing context  $\Gamma$ , the expression  $e$  has type  $C$ , where under the knowledge  $P_0$  we need to prove  $P_1$ . The knowledge  $P_0$  is our collected information that we use to prove a method correct. That means, in our typing rules, we collect the knowledge about the parameters and expressions in a method body to verify that this method body fulfills the specification defined in the method header. The verification obligation  $P_1$  should follow from the knowledge  $P_0$ .

We check if methods are well-typed with judgments of form  $Ds; Name \vdash M : OK$ . This judgment can be read as: in the definition table, the method  $M$  defined under the definition  $Name$  is correct. The typing rules of Fig. 2 are explained in the technical report [41] in detail. The first four rules type different expressions and collect the information of these expressions to prove with rule MOK that a method fulfills its specification. In the rule MOK with keyword **verify**, we call a verifier to prove each method once. Abstract methods (ABSOK) are always correct. Rule BODYOK ensures that all methods in a body are correctly typed.

### 3.3 Flattening Semantics

When we implement methods in several traits, we have to check that these traits are compatible when they are composed. This process to derive a complete class from a set of traits is called flattening. We follow the traditional flattening semantics [20]. A class that is defined by composing several traits is obtained by flattening rules. All methods are direct members of the class [20]. Overall, our flattening process works as

$Prog$	$::= Ds e$
$D$	$::= TD \mid CD$
$Name$	$::= t \mid C$
$TD$	$::= t = E$
$CD$	$::= C = E$
$E$	$::= Body \mid t \mid E + E \mid E[\mathbf{makeAbstract} m]$
$Body$	$::= \{\mathbf{interface?} [Cs] Ms\}$
$M$	$::= MH e?;$
$MH$	$::= S \mathbf{method} C m(C_1 x_1 \dots C_n x_n)$
$e$	$::= x \mid e.m(es) \mid \mathbf{new} C(es)$
$\mathcal{E}_v$	$::= [].m(es) \mid v.m(vs [] es) \mid \mathbf{new} C(vs [] es)$
$v$	$::= \mathbf{new} C(vs)$
$\Gamma$	$::= x_1 : C_1 \dots x_n : C_n$
$S$	$::= \dots e.g. \mathbf{Pre} : P \mathbf{Post} : P$
$P$	$::= \dots e.g. \text{First order logic}$

**Fig. 1.** Syntax of the trait system

a big step reduction arrow, where we reduce a trait expression into a well-typed and verified body.

To introduce our flattening rules in Fig. 3, we first define the helper functions. The function *allMeth* collects all method headers with the same name as *m* in all input bodies (Definition 1). When two *Bodys* are composed (Definition 2), the implemented interfaces are united and the methods are composed. The composition of methods (Definition 3) collects methods that are only defined in one of the input sets. If a method is in both sets, it is composed (Definition 4). Here, we distinguish four cases. If one method is abstract and the other is concrete, we have to show that the precondition of the abstract method implies the precondition of the concrete method. Additionally, the postcondition of the concrete one has to imply the postcondition of the abstract one. This is similar to Liskov’s substitution principle [34]. The second case is the symmetric variant of the first case. In the third and fourth case, two abstract methods are composed. Here, the specification of one abstract method has to imply the specification of the other abstract method such that an implementation can still satisfy all specifications of abstract methods. If both methods are concrete, the composition is correctly left undefined. This composition error can be resolved by making one method *m* abstract in the *Body*, as defined in Definition 5. The resulting *Body* is similar with the difference that the implementation of the method *m* is omitted. The flattening rules in Fig. 3 are explained in the following in detail. In these rules, a set of traits is flattened to a declaration containing all methods. If abstract and concrete methods with the same name are composed, Definitions 2–4 are used to guarantee correctness of the composition.

**Definition 1 (All Methods).**  $allMeth(m, Bodys) = \{MH; \mid Body \in Bodys, Body(m) = MH;\}$

**Definition 2 (Body Composition).**  $Body_1 + Body_2 = Body \{\mathbf{interface?} [Cs_1] Ms_1\} + \{\mathbf{interface?} [Cs_2] Ms_2\} = \{\mathbf{interface?} [Cs_1 \cup Cs_2] Ms_1 + Ms_2\}$

$$\begin{array}{c}
 \overline{\Gamma \vdash x : \Gamma(x) \dashv \text{result} : \Gamma(x) \ \& \ \text{result} = x \models \text{true}} \quad (\text{X}) \\
 \\
 \frac{
 \begin{array}{l}
 S \text{ method } C \ m(C_1 \ x_1 \dots C_n \ x_n)\_ ; \in \ \text{methods}(C_0) \\
 \Gamma \vdash e_0 : C_0 \dashv P_0 \models P'_0 \ \dots \ \Gamma \vdash e_n : C_n \dashv P_n \models P'_n \\
 x'_0 \dots x'_n \ \text{fresh} \quad S' = S[\text{this} := x'_0, x_1 := x'_1, \dots, x_n := x'_n] \\
 P = (\text{result} : C \ \& \ P_0[\text{result} := x'_0] \ \& \ \dots \ \& \ P_n[\text{result} := x'_n] \\
 \quad \& \ (\text{Pre}(S') \implies \text{Post}(S')))
 \end{array}
 }{
 \Gamma \vdash e_0.m(e_1 \dots e_n) : C \dashv P \models P'_0 \ \& \ \dots \ \& \ P'_n \ \& \ \text{Pre}(S')
 } \quad (\text{METHOD}) \\
 \\
 \frac{
 \begin{array}{l}
 \Gamma \vdash e_1 : C_1 \dashv P_1 \models P'_1 \ \dots \ \Gamma \vdash e_n : C_n \dashv P_n \models P'_n \\
 \text{getters}(C) = S_1 \ \text{method } C_1 \ x_1(); \dots S_n \ \text{method } C_n \ x_n(); \quad x'_1 \dots x'_n \ \text{fresh} \\
 S'_i = S_i[\text{this} := \text{result}] \quad P'_i = (P_i[\text{result} := x'_i] \ \& \ (\text{Pre}(S'_i) \implies \text{result}.x_i() = x'_i)) \\
 P = (\text{result} : C \ \& \ P'_1 \ \& \ \dots \ \& \ P'_n)
 \end{array}
 }{
 \Gamma \vdash \text{new } C(e_1 \dots e_n) : C \dashv P \models P'_1 \ \& \ \dots \ \& \ P'_n \ \& \ \text{Pre}(S'_1) \ \& \ \dots \ \& \ \text{Pre}(S'_n)
 } \quad (\text{NEW}) \\
 \\
 \frac{
 \Gamma \vdash e : C' \dashv P \models P' \quad C' \ \text{instance of } C
 }{
 \Gamma \vdash e : C \dashv P \models P'
 } \quad (\text{SUB}) \\
 \\
 \frac{
 \Gamma = \text{this} : \text{Name}, \ x_1 : C_1, \ \dots, \ x_n : C_n \quad \Gamma \vdash e : C \dashv P \models P' \\
 \text{verify } Ds \vdash (\Gamma \ \& \ \text{Pre}(S) \ \& \ P) \models (P' \ \& \ \text{Post}(S))
 }{
 Ds; \ \text{Name} \vdash S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n) \ e; \ : \ \text{OK}
 } \quad (\text{MOK}) \\
 \\
 \frac{
 }{
 Ds; \ \text{Name} \vdash S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n); \ : \ \text{OK}
 } \quad (\text{ABSMOK}) \\
 \\
 \frac{
 \text{Body} = \{\text{interface? } [Cs] \ M_1 \ \dots \ M_n\} \\
 Ds; \ \text{Name} \vdash M_1 : \text{OK} \ \dots \ Ds; \ \text{Name} \vdash M_n : \text{OK}
 }{
 Ds; \ \text{Name} \vdash \text{Body} \ : \ \text{OK}
 } \quad (\text{BODYTYPED})
 \end{array}$$

**Fig. 2.** Expression typing rules

**Definition 3 (Methods Composition).**  $Ms_1 + Ms_2 = Ms$

- $(M \ Ms_1) + Ms_2 = M \ (Ms_1 + Ms_2)$   
if  $\text{methName}(M) \notin \text{dom}(Ms_2)$
- $(M_1 \ Ms_1) + (M_2 \ Ms_2) = M_1 + M_2 \ (Ms_1 + Ms_2)$   
if  $\text{methName}(M_1) = \text{methName}(M_2)$
- $\emptyset + Ms = Ms$

**Definition 4 (Method Composition).**  $M_1 + M_2 = M$

- $S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n) \ e; + S' \ \text{method } C \ m(C_1 \ \dots C_n \ \_);$   
=  $S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n) \ e;$   
if  $\text{Pre}(S')$  implies  $\text{Pre}(S)$  and  $\text{Post}(S)$  implies  $\text{Post}(S')$
- $MH_1; + MH_2 \ e; = MH_2 \ e; + MH_1;$
- $S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n); + S' \ \text{method } C \ m(C_1 \ \dots C_n \ \_);$   
=  $S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n);$   
if  $\text{Pre}(S')$  implies  $\text{Pre}(S)$  and  $\text{Post}(S)$  implies  $\text{Post}(S')$
- $S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n); + S' \ \text{method } C \ m(C_1 \ \dots C_n \ \_);$   
=  $S' \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n);$   
if  $(\text{Pre}(S)$  implies  $\text{Pre}(S')$  and  $\text{Post}(S')$  implies  $\text{Post}(S))$   
and not  $(\text{Pre}(S')$  implies  $\text{Pre}(S)$  and  $\text{Post}(S)$  implies  $\text{Post}(S'))$

**Definition 5 (Body Abstraction).**  $\text{Body}[\text{makeAbstract } m]$

$$\begin{aligned}
 & \{[Cs] \ Ms_1 \ S \ \text{method } C \ m(Cxs)\_ ; \ Ms_2\}[\text{makeAbstract } m] \\
 & = \{[Cs] \ Ms_1 \ S \ \text{method } C \ m(Cxs); \ Ms_2\}
 \end{aligned}$$

$$\begin{array}{c}
\frac{D'_1 \dots D'_n \vdash D_1 \Downarrow D'_1 \quad \dots \quad D'_1 \dots D'_n \vdash D_n \Downarrow D'_n}{D_1 \dots D_n \Downarrow D'_1 \dots D'_n} \quad (\text{FLATTOP}) \\
\\
\frac{Ds; \text{Name} \vdash E \Downarrow \text{Body} \quad \text{if Name of form } C \text{ then } \text{abs}(\text{Body}) = S \ T \ x_1(); \dots \ S \ T \ x_n();}{Ds \vdash \text{Name} = E \Downarrow \text{Name} = \text{Body}} \quad (\text{DFLAT}) \\
\\
\frac{\begin{array}{l} \text{Body} = \{\text{interface? } [Cs] \ M_1 \dots M_n\} \\ \text{Body}' = \{\text{interface? } [Cs] \ M_1 \dots M_n \ Ms\} \\ Ms = \{\Sigma \text{allMeth}(Ds, \ Cs, \ m) \mid m \in \text{dom}(Cs) \text{ and } m \notin \text{dom}(\text{Body})\} \\ Ds; \text{Name} \vdash \text{Body}' : \text{OK} \end{array}}{Ds; \text{Name} \vdash \text{Body} \Downarrow \text{Body}'} \quad (\text{BFLAT}) \\
\\
\frac{}{Ds; \text{Name} \vdash t \Downarrow Ds(t)} \quad (\text{TFLAT}) \\
\\
\frac{Ds; \text{Name} \vdash E_1 \Downarrow \text{Body}_1 \quad Ds; \text{Name} \vdash E_2 \Downarrow \text{Body}_2}{Ds; \text{Name} \vdash E_1 + E_2 \Downarrow \text{Body}_1 + \text{Body}_2} \quad (+\text{FLAT}) \\
\\
\frac{Ds; \text{Name} \vdash E \Downarrow \text{Body} \quad \begin{array}{l} \text{Body} = \{[Cs] \ \overline{M}_1 \ S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n); \ \overline{M}_2\} \\ \text{Body}' = \{[Cs] \ \overline{M}_1 \ S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n); \ \overline{M}_2\} \end{array}}{Ds; \text{Name} \vdash E[\text{makeAbstract } m] \Downarrow \text{Body}'} \quad (\text{ABSFLAT})
\end{array}$$

**Fig. 3.** Flattening rules

FLATTOP. The first rule flattens a set of declarations  $D_1 \dots D_n$  to a set  $D'_1 \dots D'_n$ . We express this rule in a non-computational way: we assume to know the resulting  $D'_1 \dots D'_n$ , and we use them as a guide to compute them. Note that if there is a resulting  $D'_1 \dots D'_n$  then it is unique; flattening is a deterministic process and  $D'_1 \dots D'_n$  are used only to type check the results. They are not used to compute the shape of the flattened code.

Non computational rules like this are common with nominal type systems [27] where the type signatures of all classes and methods can be extracted before the method bodies are verified.

DFLAT. This rule flattens an individual definition by flattening the trait expression. When the flattening produces a class definition, we also check that the body denotes an instantiable class; a class whose only abstract methods are valid getters. The function  $\text{abs}(\text{Body})$  returns the abstract methods.

BFLAT. It may look surprising that the  $\text{Body}$  does not flatten to itself. This represents what happens in most programming languages, where implementing an interface implicitly imports the abstract signature for all the methods of that interface. In the context of verification also the specification of such interface methods is imported. In concrete,  $\text{Body}'$  is like  $\text{Body}$ , but we add  $Ms$  by collecting all the methods of the interfaces that are not already present in the  $\text{Body}$ .

Moreover, we check that all the methods defined in the class respect the typing and the specification defined in the interfaces: if a class has  $S \ \text{method } \text{Foo } \text{foo}();$  or  $S \ \text{method } \text{Foo } \text{foo}() \ e;$  and there is a  $S' \ \text{method } \text{Foo } \text{foo}();$  in the interface, then  $S$  must respect the specification  $S'$ . The system then checks that the  $\text{Body}$  is well-typed and verified by calling  $Ds; \text{Name} \vdash M_i : \text{OK}$

TFLAT. A trait  $t$  is flattened to its declaration  $Ds(t)$ .

+FLAT. The composition of two expression  $E_1$  and  $E_2$ , where both expressions are first reduced to  $Body_1$  and  $Body_2$ , results in the composition of these bodies as defined in Definition 2.

ABSFLAT. An expression  $E$  where one method  $m$  is made abstract flattens to a  $Body'$ . We know that  $E$  flattens to  $Body$ . The only difference between  $Body$  and  $Body'$  is that the one method  $m$  is abstract in  $Body'$ . In  $Body$ , the method can be abstract or concrete.

### 3.4 Soundness of the Trait-based CbC Process

In this section, we formulate our main result of the TraitCbC process. We prove soundness of the flattening process with a parametric logic. The proofs of the lemmas and theorems are in the technical report [41]. We claim that if you have a language without code reuse and with sound and modular PhV verification then the language supports CbC simply by adding traits to the language. That is, traits intrinsically enable a CbC program construction process.

To prove soundness of the refinement process of TraitCbC (Theorem 2: Sound CbC Process) as exemplified in Sect. 2, we have to show that the flattening process is correct (Theorem 1: General Soundness). In turn, to prove General Soundness, we need two lemmas which state that the composition of traits is correct (Lemma 1) and that a trait after the `makeAbstract` operation is still correct (Lemma 2).

In Lemma 1, we have well-typed definitions  $Ds$ , and two well-typed and verified traits in  $Ds$ , and the resulting trait/class is also well-typed and verified.

#### Lemma 1 (Composition correct).

If  $Ds(t_1) = Body_1$ ,  $Ds(t_2) = Body_2$ ,  $Ds(Name) = Body$ ,  $Ds; t_1 \vdash Body_1 : OK$ ,  
 $Ds; t_2 \vdash Body_2 : OK$ , and  $Body_1 + Body_2 = Body$ ,  
 then  $Ds; Name \vdash Body : OK$

Lemma 2 shows that if we have a well-typed and verified trait, the operation `makeAbstract` results in a trait/class that is also well-typed and verified.

#### Lemma 2 (MakeAbstract correct).

If  $Ds(t) = Body$ ,  $Ds(Name) = Body'$ ,  $Ds; t \vdash Body : OK$ ,  
 and  $Body[\text{makeAbstract } m] = Body'$ ,  
 then  $Ds; Name \vdash Body' : OK$

With these Lemmas, we can prove Theorem 1. Given a sound and modular verification language, then all programs that flatten are well-typed and verified. In a modular verification language, a method can be fully verified using only the information contained in the method declaration and the specification of any used method. Moreover, our parametric logic must support at least a commutative and associative *and* (but of course other ways to merge knowledge could work too) and a transitive *implication* (but of course other forms of logical consequence could work too).

#### Theorem 1 (General Soundness).

For all programs  $Ds$  where  $Ds$  flattens to  $Ds'$ , and  $Ds'$  is well-typed;  
 that is, for all  $Name = Body \in Ds'$ , we have  $Ds'; Name \vdash Body : OK$ .

We now show that the TraitCbC process is sound. Theorem 2 states that starting with one abstract method and a set of verified traits, the composed program is also verified.

**Theorem 2 (Sound CbC Process).**

Starting from a fully abstract specification  $t_0$ , and some refinement steps  $t_1 \dots t_n$ , we can write  $C = t_0 + \dots + t_n$  as our whole CbC refinement process; where  $t_0 + t_1$  is the application of the first refinement step. If we use CbC to construct programs, we can start from verified atomic units and get a verified result. Formally, if  $t_0 = \{MH\}$   $t_1 = \{Ms_1\}$  ...  $t_n = \{Ms_n\}$  are well-typed, and

$$\begin{array}{lcl} t_0 = \{MH\} & & t_0 = \{MH\} \\ t_1 = \{Ms_1\} \dots t_n = \{Ms_n\} & \Downarrow & t_1 = \{Ms_1\} \dots t_n = \{Ms_n\} \\ C = t_0 + \dots + t_n & & C = \text{Body} \end{array}$$

then  $C = \text{Body}$  is well-typed.

*Proof.* This is a special case of Theorem 1.

Theorem 2 shows clearly that trait composition intrinsically enables a CbC refinement process: A object-oriented programming language with traits and a corresponding specification language supports an incremental CbC approach.

**Table 1.** Comparison of TraitCbC with classical CbC

	Classic CbC	TraitCbC
Language	Additional rules for a programming language.	Programming language with traits. Needs specification language.
Tool support	Pen and paper. Some specialized tools available.	Relies on prevalent PhV verification tools.
Construction Rules	Specific refinement rules.	Refinement by composition of traits.
Debugging	Guarantees the correctness of each refinement step. Only refinements without abstract statement are directly verified.	Guarantees the correctness of each refinement step. Each method is specified such that each refinement can directly be verified.
Proof complexity	Many, but small proofs .	Any granularity of proofs.
Reuse	Refinement steps cannot be reused; only fully implemented methods can.	Each verified method in a trait can be reused.
Applications	Focuses on small but correctness-critical algorithms.	As TraitCbC is based on PhV, it can be used in areas of PhV. Additionally, traits are beneficial for incremental development approaches and development of software product lines.



## 4 Trait-Based Correctness-by-Construction in Comparison to Classical CbC

In this section, we discuss the benefits of TraitCbC in comparison to classical CbC. To do this, we describe classical CbC first.

Classical correctness-by-construction (CbC) [19,30,37] is an incremental approach to construct programs. CbC uses a *Hoare triple* specification  $\{P\} S \{Q\}$  stating that if the precondition  $P$  holds, and the statement  $S$  is executed, then the statement terminates and postcondition  $Q$  holds. The CbC refinement process starts with a Hoare triple where the statement  $S$  is abstract. This abstract statement can be seen as a hole in the program that needs to be filled. With a set of refinement rules, an abstract statement is replaced by more concrete statements (i.e., statements in the guarded command language [19] that can contain further abstract statements). The process stops, when all abstract statements are refined to concrete statements so that no holes remain in the program. As each refinement rule is sound and each correct application of a refinement rule guarantees to satisfy the starting Hoare triple, the resulting program is correct-by-construction [30]. The CbC process is strictly tied to a set of predefined refinement rules. A programmer cannot deviate from this concept. To apply a refinement rule, it has to be checked that conditions of the rule application are satisfied. This is done by pen-and-paper or with specialized tools [42].

In Table 1, we compare TraitCbC and classical CbC:

*Language.* The classical CbC approach is external to a programming language. It needs the definition of refinement rules. TraitCbC is usable with languages that have traits, a specification language, and a corresponding verification framework. In this work, we focus on object-orientation, but the general TraitCbC programming guideline presented in this paper is also suitable for functional programming environments using abstract and concrete functions with specifications instead of traits and methods.

*Tool Support.* To use one of the approaches, tool support is desired. For classical CbC, mostly pen and paper is used. There are a few specialized tools such as CorC [42], tool support for ArcAngel [38], and SOCOS [4,5]. These tools force a certain programming procedure on the user. This procedure can be in conflict with their preferred programming style. For TraitCbC, tools for post-hoc verification can be reused. There are tools for many languages such as Java [3], C [16], C# [7,8]. Other languages are integrated with their verifier from the start, e.g., Spec# [8] and Dafny [32]. TraitCbC as presented in this paper is a core calculus, designed to show the feasibility of the concept. We believe that scaling up TraitCbC to a complete programming language reusing existing verification techniques would be feasible and would result in a similarly expressive verification process, but supporting more flexible program composition. In Sect. 5, we show how a prototype can be constructed by using the KeY verifier [3].

*Construction Rules.* To construct a program, classical CbC has a strict concept of refinement rules. A programmer cannot deviate from the granularity of the rules. In contrast, PhV does not give a mandatory guideline how to construct programs. TraitCbC is a bridge between both extremes. Programs can be constructed stepwise as with classical CbC, but if desired, any number of refinement steps can be condensed up to PhV based programming.

*Debugging.* If errors occur in the development process, TraitCbC gives early and detailed information. By specifying the method under development and any abstract

method that is called by this method, we can directly verify the correctness of the method under development. We assume that the introduced abstract methods will be correctly implemented in further refinement steps. With each step, the programmer gets closer to the solution until finally all abstract methods are implemented. Classical CbC relies on the same process, but here the abstract statements (similar to our abstract methods) are not explicitly specified by the programmer. Additional specifications in classical CbC are introduced only with some rules such as an intermediate condition in the composition rule. Then, these specifications are propagated through the program to be constructed. When arriving at a leaf in the refinement process, the correctness of the statement can be guaranteed. The problem in classical CbC is that all refinement steps where abstract statements occur cannot be verified directly. In the worst case, a wrong specification is found only after a few refinement steps.

*Proof Complexity.* TraitCbC can have the same granularity and also the same proof effort as classical CbC, since each method implementation can correspond to just one refinement step. The advantage of TraitCbC is that programmers can freely implement a method body. They must not stick to the same granularity as in the classical CbC refinement rules. As in PhV, they can implement a complete method in one step. The programmer can balance proof complexity against verifier calls.

*Reuse.* If we want to reuse developed methods or refinement steps, the approaches differ. In classical CbC, no refinement steps can be reused. A fully refined method can be reused in both approaches. For TraitCbC, we can easily reuse even very small units of code, since they are represented as methods in the traits.

*Applications.* The classical CbC approach does not scale well to development procedures for complete software system. Rather, individual algorithms can be developed with CbC [50]. As soon as we scale TraitCbC to real languages, we have the same application scenarios as PhV. As argued by Damiani et al. [18] traits enable an incremental process of specifying and verifying software. Bettini et al. [10] proposed to use traits for software product line development and highlighted the benefits of fine-grained reuse mechanisms. Here, TraitCbC's guideline is suitable for constructing new product lines step by step from the beginning.

*Summary.* In summary, TraitCbC bridges the gap between PhV and CbC. It enables a CbC process for trait-based languages without introducing refinement rules. The concrete realization of specifying and verifying methods is similar to PhV, but additionally to PhV, TraitCbC provides an incremental development process. This development process combined with the flexibility of traits allows correct methods to be developed in small and reusable steps. Moreover, we have introduced a core calculus and proved that the construction and composition of trait-based programs is correct.

## 5 Proof-of-Concept Implementation

In this section, we describe the implementation, which instantiates TraitCbC in Java with JML [31] as specification language and KeY [3] as verifier for Java code. Our trait implementation is based on interfaces with default implementation. Our open source tool is implemented in Java and integrated as plug-in in the Eclipse IDE.<sup>4</sup>

---

<sup>4</sup> Tool and evaluation at <https://github.com/TUBS-ISF/CorC/tree/TraitCbC>.

Besides this prototype, other languages with a suitable verifier, such as Dafny [32] and OpenJML [17], can also be used to implement TraitCbC.

In Listing 1, we show the concrete syntax. Each method in a trait is specified with JML with the keywords `requires` and `ensures` for the pre- and postcondition. To verify the correctness of programs, we need two steps. First, we verify the correctness of a method implemented in a trait w.r.t. its specification. Second, for trait composition, our implementation checks the correct composition for all methods (cf. Definition 2). It is verified that the specification of a concrete method satisfies the specification of the abstract one with the same signature (cf. Definition 4). These verification goals are sent to KeY, which starts an automatic verification attempt. The syntax of trait composition is shown in line 24. In a separate tc-file, the name of the resulting trait is given and the composed traits are connected with a plus operator.

```

1  public interface MaxElement1 {
2  /*@ requires list.size() > 0;
3     @ ensures (\forall int n; list.contains(n);
4     @ \result >= n) & list.contains(\result);
5     @*/
6     public default int maxElement(List list) {
7         if (list.size() == 1) return accessHead(list);
8         if (list.element() >= maxElement(list.tail()))
9             { return accessHead(list) }
10        else { return maxTail(list) } }
11
12 /*@ requires list.size() > 0;
13     @ ensures \result == list.element();
14     @*/
15     public int accessHead(List list);
16
17 /*@ requires list.size() > 1;
18     @ ensures (\forall int n; list.tail().contains(n);
19     @ \result >= n) & list.tail().contains(\result);
20     @*/
21     public int maxTail(List list);
22 }
23
24 ComposedMax = MaxElement1 + MaxElement2

```

**Listing 1.** Example in our implementation

*Evaluation.* We evaluate our implementation by a feasibility study. First, we reimplemented an already verified case study in our trait-based language. We used the IntList [43] case study, which is a small software product line (SPL) with a common code base and several features extending this code base. Here, we can show that our trait-based language also facilitates reuse. The IntList case study implements functionality to insert integers to a list in the base version. Extensions are the sorting of the list and different insert options (e.g., front /back). We implement five methods that exists in different variants with our trait-based CbC approach. We implement the case study in different granularities. The coarse-grained version is similar to the SPL implementation we started with [43], confirming that traits are also amenable to implement SPLs as shown by Bettini et al. [10]. The fine-grained version implements the five methods incrementally with 12 refinement steps. We can reuse 6 of these steps during the construction of method variants.

We also implement three more case studies BankAccount [48], Email [25], and Elevator [39] with TraitCbC and CbC to show that it is feasible to implement object-oriented programs with both approaches. We used CorC [42] as an instance of a CbC tool. We were able to implement nine classes and verify 34 methods with a size of 1–20 lines of code. For future work, a user study is necessary to evaluate the usability of TraitCbC in comparison to CbC to confirm our stated advantages.

## 6 Related Work

Traits are introduced in many languages to support clean design and reuse, for example Smalltalk [20], Java [12] by utilizing default methods in interfaces, and other Java-like languages [11, 33, 45]. The trait language TraitRecordJ was extended to support post-hoc verification of traits [18]. The authors added specifications of methods in traits for the verification of correct trait composition and proposed a modular and incremental verification process. None of these trait languages were used to formulate a refinement process to create correct programs. They only focus on code reuse or post-hoc verification.

Automatic verification is widely used for different programming languages. The object-oriented language Eiffel focuses on design-by-contract [35, 36]. All methods in classes are specified with pre-/postconditions and invariants for verification purposes. The tool AutoProof [29, 49] is used to verify the correctness of implemented methods. It translates methods to logic formulas, and an SMT solver proves the correctness. For C#, programs written in the similar language Spec# [8] are verified with Boogie. That is, code and specification are translated to an intermediate language and verified [7]. For C, the tool VCC [16] reuses the Spec# tool chain to verify programs. The tool VeriFast [28] is able to verify C and Java programs specified in separation logic. For Java, KeY [3] and OpenJML [17] verify programs specified with JML. TraitCbC is parametric in the specification language, meaning that a trait-based language with a specification language and a corresponding program verifier can be used to instantiate TraitCbC. In our implementation, we use KeY [3] to prove the correctness of methods and trait composition.

Event-B [1] is a related correctness-by-construction approach. In Event-B, automata-based systems are specified and refined to a concrete implementation. Event-B is implemented in the Rodin platform [2]. In comparison to CbC by Kourie and Watson [30] as used in this paper, Event-B works on a different abstraction level with automata-based systems instead of program code. The CbC approaches of Back et al. [6] and Morgan [37] are also related. Back et al. [6] start with explicit invariants and pre-/postconditions to refine an abstract program to a concrete implementation, while Kourie and Watson only start with a pre-/postcondition specification. These refinement approaches use specific refinement rules to construct programs which are external to the programming language. With TraitCbC, we propose a refinement procedure that is part of the language by using trait composition.

Abstract execution [47] verifies the correctness of methods with abstract, but formally specified expressions. Abstract Execution is similar to our refinement procedure where abstract methods are called in methods under construction. The difference is that abstract execution extends a programming language to use any expression in the abstract part, not only method calls. Therefore, abstract execution can better reason about irregular termination (e.g., break/continue) of methods. In comparison to

TraitCbC, abstract execution is a verification-centric approach without a guideline on how to construct programs.

Synthesis of function summaries is also related [14,26,44]. Here, verification tools automatically synthesize pre-/postconditions from functions to achieve modular verification and speed up the verification time. In comparison, TraitCbC is a complete software development approach where specification and code are developed simultaneously by a developer to achieve a correct solution. Function summaries are just a verification technique.

## 7 Conclusion

In this work, we present TraitCbC that guides programmers to correct implementations. In comparison to classical CbC, TraitCbC uses method calls and trait composition instead of refinement rules to guarantee functional correctness. We formalize the concept of a trait-based object-oriented language where the specification language is parametric to allow a broader range of languages to adopt this concept. The main advantage of TraitCbC is the simplicity of the refinement process that supports code and proof reuse.

As future work, we want to investigate how TraitCbC can be used to construct software product lines. As proposed by Bettini et al. [10], trait languages are able to implement SPLs. We want to extend the guideline of TraitCbC to construct SPLs with a refinement-based procedure that guarantees the correctness of the whole SPL. To reduce specification effort in TraitCbC, inheritance of traits is useful. Another option is to integrate the concept of Rebêlo et al. [40] which supports the design-by-contract approach with AspectJML and integrates crosscutting contract modularization to reduce redundant specifications.

Since TraitCbC is parametric in the specification logic, TraitCbC's soundness only holds if such logic is consistent when composed in the presented manner. In particular, the logic needs to take into account ill-founded specifications and non-terminating recursion. In verification, ill-founded specifications and termination issues are often considered as a second step<sup>5</sup>, separately from the verification of individual methods, and our prototype still does not yet take care of this second step. That means that methods are verified under the assumption that all other methods respect their contracts. If ill-founded specifications and non-terminating recursion are handled naively, verification might be unsound because of ill-founded reasoning. The technical report [41] shows that this problem is even more pervasive in the case of trait composition or any other form of multiple inheritance: naive composition of correct traits may produce incorrect results.

## References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-B. *STTT* **12**(6), 447–466 (2010)

---

<sup>5</sup> For example, Dafny approximately checks that the functions used in a specification form an acyclic graph.



3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: *Deductive Software Verification-The KeY Book: From Theory to Practice*, vol. 10001. Springer, Cham (2016). <https://doi.org/10.1007/978-3-319-49812-6>
4. Back, R.J.: Invariant based programming: basic approach and teaching experiences. *FAOC* **21**(3), 227–244 (2009)
5. Back, R.-J., Eriksson, J., Myreen, M.: Testing and verifying invariant based programs in the SOCOS environment. In: Gurevich, Y., Meyer, B. (eds.) *TAP 2007*. LNCS, vol. 4454, pp. 61–78. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-73770-4\\_4](https://doi.org/10.1007/978-3-540-73770-4_4)
6. Back, R.J., Wright, J.: *Refinement Calculus: A Systematic Introduction*. Springer, New York (2012). <https://doi.org/10.1007/978-1-4612-1674-2>
7. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the spec# experience. *Commun. ACM* **54**(6), 81–91 (2011)
8. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) *CASSIS 2004*. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-30569-9\\_3](https://doi.org/10.1007/978-3-540-30569-9_3)
9. ter Beek, M.H., Cleophas, L., Schaefer, I., Watson, B.W.: X-by-construction. In: Margaria, T., Steffen, B. (eds.) *ISoLA 2018*. LNCS, vol. 11244, pp. 359–364. Springer, Cham (2018). [https://doi.org/10.1007/978-3-030-03418-4\\_21](https://doi.org/10.1007/978-3-030-03418-4_21)
10. Bettini, L., Damiani, F., Schaefer, I.: Implementing software product lines using traits. In: *SAC*, pp. 2096–2102 (2010)
11. Bettini, L., Damiani, F., Schaefer, I., Strocchio, F.: TRAITRECORDJ: a programming language with traits and records. *Sci. Comput. Program.* **78**(5), 521–541 (2013)
12. Bono, V., Mensa, E., Naddeo, M.: Trait-oriented programming in Java 8. In: *PPPJ*, pp. 181–186 (2014)
13. Chapman, R.: Correctness by construction: a manifesto for high integrity software. In: *SCS*, pp. 43–46 (2006)
14. Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wachter, B.: Synthesising interprocedural bit-precise termination proofs (t). In: *ASE*, pp. 53–64 (2015)
15. Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley (2002)
16. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03359-9\\_2](https://doi.org/10.1007/978-3-642-03359-9_2)
17. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) *NFM 2011*. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20398-5\\_35](https://doi.org/10.1007/978-3-642-20398-5_35)
18. Damiani, F., Dovland, J., Johnsen, E.B., Schaefer, I.: Verifying traits: an incremental proof system for fine-grained reuse. *FAOC* **26**(4), 761–793 (2014)
19. Dijkstra, E.W.: *A Discipline of Programming*. Prentice Hall (1976)
20. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: a mechanism for fine-grained reuse. *TOPLAS* **28**(2), 331–388 (2006)
21. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and Mixins. In: *POPL*, pp. 171–183 (1998)
22. Gries, D.: *The Science of Programming*. Springer, New York (1987). <https://doi.org/10.1007/978-1-4612-5983-1>

23. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 100–115. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-39634-2\\_10](https://doi.org/10.1007/978-3-642-39634-2_10)
24. Hall, A., Chapman, R.: Correctness by construction: developing a commercial secure system. *Softw. IEEE* **19**(1), 18–25 (2002)
25. Hall, R.J.: Fundamental nonmodularity in electronic mail. *ASE* **12**(1), 41–79 (2005)
26. Hoare, C.A.R.: Procedures and parameters: an axiomatic approach. In: Engeler, E. (ed.) *Symposium on Semantics of Algorithmic Languages*. LNM, vol. 188, pp. 102–116. Springer, Heidelberg (1971). <https://doi.org/10.1007/BFb0059696>
27. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS* **23**(3), 396–450 (2001)
28. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) *APLAS 2010*. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-17164-2\\_21](https://doi.org/10.1007/978-3-642-17164-2_21)
29. Khazeev, M., Rivera, V., Mazzara, M., Johard, L.: Initial steps towards assessing the usability of a verification tool. In: Ciancarini, P., Litvinov, S., Messina, A., Sillitti, A., Succi, G. (eds.) *SEDA 2016*. AISC, vol. 717, pp. 31–40. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-70578-1\\_4](https://doi.org/10.1007/978-3-319-70578-1_4)
30. Kourie, D.G., Watson, B.W.: *The Correctness-by-Construction Approach to Programming*. Springer, Heidelberg (2012). <https://doi.org/10.1007/978-3-642-27919-5>
31. Leavens, G.T., Baker, A.L., Ruby, C.: JML: a Java modeling language. In: *Formal Underpinnings of Java Workshop (at OOPSLA 1998)*, pp. 404–420. Citeseer (1998)
32. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) *LPAR 2010*. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-17511-4\\_20](https://doi.org/10.1007/978-3-642-17511-4_20)
33. Liquori, L., Spiwack, A.: FeatherTrait: a modest extension of featherweight Java. *TOPLAS* **30**(2), 1–32 (2008)
34. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. *TOPLAS* **16**(6), 1811–1841 (1994)
35. Meyer, B.: Eiffel: a language and environment for software engineering. *JSS* **8**(3), 199–246 (1988)
36. Meyer, B.: Applying “Design by Contract”. *Computer* **25**(10), 40–51 (1992)
37. Morgan, C.: *Programming from Specifications*, 2nd edn. Prentice Hall (1994)
38. Oliveira, M.V.M., Cavalcanti, A., Woodcock, J.: ArcAngel: a tactic language for refinement. *FAOC* **15**(1), 28–47 (2003)
39. Plath, M., Ryan, M.: Feature integration using a feature construct. *Sci. Comput. Program.* **41**(1), 53–84 (2001)
40. Rebêlo, H., et al.: AspectJML: modular specification and runtime checking for crosscutting contracts. In: *MODULARITY*, pp. 157–168. ACM, New York (2014)
41. Runge, T., Potanin, A., Thüm, T., Schaefer, I.: Traits for correct-by-construction programming (2022). <https://arxiv.org/abs/2204.05644>
42. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool support for correctness-by-construction. In: Hähnle, R., van der Aalst, W. (eds.) *FASE 2019*. LNCS, vol. 11424, pp. 25–42. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2)
43. Scholz, W., Thüm, T., Apel, S., Lengauer, C.: Automatic detection of feature interactions using the Java modeling language: an experience report. In: *SPLC*. ACM, New York (2011)



44. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based function summaries in bounded model checking. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 160–175. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-34188-5\\_15](https://doi.org/10.1007/978-3-642-34188-5_15)
45. Smith, C., Drossopoulou, S.: *Chai*: traits for Java-like languages. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 453–478. Springer, Heidelberg (2005). [https://doi.org/10.1007/11531142\\_20](https://doi.org/10.1007/11531142_20)
46. Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-71067-7\\_23](https://doi.org/10.1007/978-3-540-71067-7_23)
47. Steinhöfel, D., Hähnle, R.: Abstract execution. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 319–336. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-30942-8\\_20](https://doi.org/10.1007/978-3-030-30942-8_20)
48. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-based deductive verification of software product lines. In: GPCE, pp. 11–20. ACM (2012)
49. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46681-0\\_53](https://doi.org/10.1007/978-3-662-46681-0_53)
50. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.: Correctness-by-construction and post-hoc verification: a marriage of convenience? In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 730–748. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-47166-2\\_52](https://doi.org/10.1007/978-3-319-47166-2_52)

## **A.5. Flexible Correct-by-Construction Programming**

---

## FLEXIBLE CORRECT-BY-CONSTRUCTION PROGRAMMING

TOBIAS RUNGE<sup>a,b</sup>, TABEA BORDIS<sup>a,b</sup>, ALEX POTANIN<sup>d</sup>, THOMAS THÜM<sup>e</sup>,  
AND INA SCHAEFER<sup>a,b,c</sup>

<sup>a</sup> Institute of Information Security and Dependability (KASTEL), Karlsruhe Institute of Technology, Germany

*e-mail address*: {tobias.runge, tabea.bordis, ina.schaefer}@kit.edu

<sup>b</sup> Institute of Software Engineering and Automotive Informatics, TU Braunschweig, Germany

<sup>c</sup> School for Data-Science and Computational Thinking, Stellenbosch University, South Africa

<sup>d</sup> School of Computing, Australian National University, Australia

*e-mail address*: alex.potantin@anu.edu.au

<sup>e</sup> Institute of Software Engineering and Programming Languages, University of Ulm, Germany

*e-mail address*: thomas.thuem@uni-ulm.de

---

ABSTRACT. Correctness-by-Construction (CbC) is an incremental program construction process to construct functionally correct programs. The programs are constructed stepwise along with a specification that is inherently guaranteed to be satisfied. CbC is complex to use without specialized tool support, since it needs a set of predefined refinement rules of fixed granularity which are additional rules on top of the programming language. Each refinement rule introduces a specific programming statement and developers cannot depart from these rules to construct programs. CbC allows to develop software in a structured and incremental way to ensure correctness, but the limited flexibility is a disadvantage of CbC. In this work, we compare classic CbC with CBC-BLOCK and TRAITCBC. Both approaches CBC-BLOCK and TRAITCBC, are related to CbC, but they have new language constructs that enable a more flexible software construction approach. We provide for both approaches a programming guideline, which similar to CbC, leads to well-structured programs. CBC-BLOCK extends CbC by adding a refinement rule to insert any block of statements. Therefore, we introduce CBC-BLOCK as an extension of CbC. TRAITCBC implements correctness-by-construction on the basis of traits with specified methods. We formally introduce TRAITCBC and prove soundness of the construction strategy. All three development approaches are qualitatively compared regarding their programming constructs, tool support, and usability to assess which is best suited for certain tasks and developers.

---

*Key words and phrases*: Traits, Correctness-by-Construction, Formal Methods, Post-hoc Verification.

## 1. INTRODUCTION

*Correctness-by-Construction* (CbC) [Dij76, Gri87, KW12, Mor94] is a methodology in the field of formal methods to incrementally construct functionally correct programs guided by a pre-/postcondition specification.<sup>1</sup> In contrast to post-hoc verification, where a program is typically specified and verified after implementing it, CbC is based around successively creating a program together with the specification. This is achieved by applying refinement rules from a small set of defined rules where in each refinement step, an abstract statement (i.e., a hole in the program) is refined to a more concrete implementation that can still contain some nested abstract statements. While refining the program, the correctness of the whole program is guaranteed through applicability conditions that are defined in the refinement rules. The construction ends when no abstract statement is left.

The underlying idea of this specification-first, refinement-based approach is that developers are forced to think about their algorithm more thoroughly rather than having a trial-and-error verification approach. This trial-and-error verification can oftentimes be experienced with post-hoc verification because programs are implemented first and therefore not well-structured for the verification process which leads to tedious verification work. Additionally, through the structured reasoning discipline that is enforced by the refinement rules in CbC, errors are more likely to be detected earlier in the design process, and it is argued that program quality increases and verification effort is reduced [KW12, WKSC16].

Despite these benefits, CbC intuitively has a drawback: The flexibility of creating a program is limited to the set of refinement rules and the rigid, rule-based construction process of applying one rule at a time. This is even increased by the granularity of the rules which explicitly only allow to use one language construct at a time (e.g., one assignment to a variable). Additionally, the refinement rules extend the programming language (i.e., refinement rules are an additional linguistic construct to transform programs), and therefore special tool support (e.g., CORC [RSC<sup>+</sup>19, BCK<sup>+</sup>22]) is necessary to introduce the CbC refinement process to a programming language. As a result, the barrier to construct programs using CbC is large because the approach at first seems unintuitive and requires effort, knowledge, and special tool support.

In this article, we introduce two alternative correctness-by-construction development approaches that relax the inflexible CbC construction approach without losing the benefits of CbC itself. Both introduce more flexible language constructs to create programs which allow to condense construction steps that tackle the complex and strict programming style of CbC. The goal is to propose a usable CbC approach that offers reasonable constructs to develop programs correctly. Therefore, we qualitatively discuss our two proposed approaches and the original CbC approach regarding their programming constructs, tool support, and usability to assess their benefits and drawbacks.

First, we present CBC-BLOCK which adds new refinement rules. This introduction of new refinement rules should not be seen as a further restriction, but as a relaxation of the procedure. These new refinement rules increase the ways in which programs can be developed as they allow to refine abstract statements to a specified block of code that fulfills its specification. This basically means that this block can contain multiple assignments,

---

<sup>1</sup>The approach should not be confused with other CbC approaches such as CbyC of Hall and Chapman [HC02]. CbyC is a software development process that uses formal modeling techniques and analysis for various stages of development (architectural design, detailed design, code) to detect and eliminate defects as early as possible [Cha06]. We also exclude data refinement from abstract data types to concrete ones during code generation as for example in Isabelle/HOL [HKKN13].

selections, or loops whereas with classic CbC for each assignment, selection, and loop a new refinement step is needed. Initially, a block is just an abstract placeholder, but it has a pre-/postcondition specification so that the introduced specification of the block can be checked against the specification of the refined abstract statement. In a next step, the block is instantiated by some code, and it is directly proved that this code fulfills its own specification. The idea of the block rules is similar to a method call, but a block can alter local variables in the method under construction. A block of code can contain further blocks which can be subsequently refined. Consequently, any nesting of blocks may occur. CBC-BLOCK is implemented as extension of the CORC tool support.

Second, we present TRAITCBC which is a new software development approach that enables correct-by-construction development by method abstraction and method composition without relying on refinement rules and special tool support. TRAITCBC uses *traits* [DNS<sup>+</sup>06], which are a flexible object-oriented language construct supporting a rich form of modular code reuse orthogonal to inheritance. A trait is a set of *concrete* or *abstract* methods (i.e., the method has either a body or has no body).<sup>2</sup> Traits can be composed into a larger trait or into a class that contains all methods of all composed traits. Trait composition exists as a direct concept of the programming language [DNS<sup>+</sup>06] instead of being a program transformation concept, such as the CbC refinement rules. On the basis of traits, TRAITCBC introduces a programming guideline for an incremental program construction approach that guarantees that the resulting program is correct by construction. A construction step comprises the development of a method and direct composition with the existing code base to ensure correctness. TRAITCBC allows the implementation of any method size and complexity as long as the methods are composable with respect to their specification. Even with this flexibility, TRAITCBC keeps the advantages of a structured incremental development approach.

The contribution of our article is to demonstrate and compare the range of possibilities to develop programs correct by construction from strict rule-based CbC to the more relaxed CBC-BLOCK to TRAITCBC without any refinement rules. In this article, we introduce TRAITCBC and explain the typing, reduction, and flattening rules. We give a proof that TRAITCBC guarantees to develop programs correct by construction. We also present the CBC-BLOCK approach with the block refinement rules. All approaches are implemented in the CORC [RSC<sup>+</sup>19] tool support. We compare and discuss classic CbC, CBC-BLOCK, and TRAITCBC qualitatively to assess their benefits and drawbacks. This article extends previous work [RPTS22] by introducing the typing and reduction rules of TRAITCBC in detail. The soundness proof of TRAITCBC is also presented in this article. CBC-BLOCK is a new approach that has not been presented before.

## 2. CORRECTNESS-BY-CONSTRUCTION

Classic correctness-by-construction (CbC) [Dij76, KW12, Mor94] is an incremental approach to construct programs. CbC uses a *Hoare triple* specification  $\{P\} S \{Q\}$  stating that if the precondition  $P$  holds, and the statement  $S$  is executed, then the statement terminates and postcondition  $Q$  holds. The CbC refinement process starts with a Hoare triple where the statement  $S$  is abstract. This abstract statement can be seen as a hole in the program that needs to be filled. With a set of refinement rules, an abstract statement is replaced by more

<sup>2</sup>Java interfaces with default methods are a good approximation for what has been called trait in the literature

Definition 1: Refinement Rules for the Correctness-by-Construction Approach

Let  $P$  be the precondition,  $Q$  be the postcondition, and  $S$  be an abstract statement. Then, the Hoare triple  $\{P\}S\{Q\}$  is **refinable** to

- **Skip:**  $\{P\}\text{skip}\{Q\}$  iff  $P$  implies  $Q$
- **Assignment:**  $\{P\} \mathbf{x} := E\{Q\}$  iff  $P$  implies  $Q[\mathbf{x} := E]$
- **Composition:**  $\{P\}S_1; S_2\{Q\}$  iff intermediate condition  $M$  exists such that  $\{P\}S_1\{M\}$  and  $\{M\}S_2\{Q\}$  hold
- **Selection:**  $\{P\}\text{if } G_1 \rightarrow S_1 \text{ elseif } \dots G_n \rightarrow S_n \text{ fi}\{Q\}$  iff  $P$  implies  $G_1 \vee \dots \vee G_n$  and  $\forall i \in \{1 \dots n\} : \{P \wedge G_i\}S_i\{Q\}$  holds
- **Repetition:**  $\{P\}\text{do } [I, V] G \rightarrow S \text{ od}\{Q\}$  iff  $P$  implies  $I$  and  $I \wedge \neg G$  implies  $Q$  and  $\{I \wedge G\}S\{I\}$  holds and  $\{I \wedge G \wedge V = V_0\}S\{I \wedge 0 \leq V < V_0\}$  holds
- **Weaken precondition:**  $\{P'\}S\{Q\}$  iff  $P$  implies  $P'$
- **Strengthen postcondition:**  $\{P\}S\{Q'\}$  iff  $Q'$  implies  $Q$
- **Method Call:**  $\{P\}\mathbf{m}(a_1, \dots, a_n) \rightarrow b\{Q\}$  iff method  $\{P'\}\mathbf{m}(p_1, \dots, p_n) \rightarrow r\{Q'\}$  exists and  $P$  implies  $P'[p_i \setminus a_i]$  and  $Q'[\text{old}(p_i) \setminus \text{old}(a_i), r \setminus b]$  implies  $Q$

[RSC<sup>+</sup>19, KW12]

concrete statements (i.e., statements in the guarded command language [Dij76] that can contain further abstract statements). The process stops, when all abstract statements are refined to concrete statements so that no holes remain in the program. As each refinement rule is sound and each correct application of a refinement rule guarantees to satisfy the starting Hoare triple, the resulting program is correct by construction [KW12]. The CbC approach is strictly tied to this set of predefined refinement rules. A developer cannot deviate from this concept.

In Definition 1, we present the eight refinement rules of CbC by Kourie and Watson [KW12]. The concrete program statements are written in the guarded command language [Dij75]. To apply a refinement rule, it has to be checked that side conditions of the rule application are satisfied. This is done by pen-and-paper or with specialized tools [RSC<sup>+</sup>19]. For example, the skip rule introduces an empty statement that does not alter the program state. This refinement is applicable if and only if the precondition  $P$  implies the postcondition  $Q$ . The composition rule splits the Hoare triple  $\{P\}S\{Q\}$  into two Hoare triples by using an intermediate condition  $M$ . This refinement is applicable, if and only if the two new Hoare triples are correct. Of course, the statements  $S_1$  and  $S_2$  are still abstract and can be further refined.

### 3. CBC-BLOCK— CBC WITH BLOCK CONTRACTS

In this section, we introduce the CBC-BLOCK approach that adds two new refinement rules to classic CbC. The new refinement rules increase the ways to construct programs. Therefore, the rigid CbC approach is loosened while retaining the benefits of a structured program construction approach. A block rule refines an abstract statement to a block that is specified with a block contract (i.e., a pre-/postcondition specification for that block) [ABB<sup>+</sup>16]. The block is a special statement that can be further refined in two ways. Similar to an abstract statement, any CbC refinement rules can be applied. Additionally, the block can

be instantiated by any sequence of concrete statements and further blocks with a block-instantiation rule. Thus, a block can be used to condense the application of several CbC refinement rules. For example, a block can be instantiated with a while-loop that already contains a concrete body. This instantiation replaces the application of the repetition rule and at least one assignment rule. We introduce `CBC-BLOCK` with a motivating example and present the block rules to introduce and instantiate blocks. In the end of this section, we present `CBC-BLOCK` implemented in the CbC tool `CORC`<sup>3</sup> and evaluate its usability with a user study.

**3.1. Motivating Example.** In this section, we exemplify the `CBC-BLOCK` approach by implementing a `maxElement` algorithm. The `maxElement` algorithm searches the largest element in a list of integers. The list supports a `get`-method which returns the element at the specified position in this list. A `contains`-method checks that the result is a member of the list. We iterate with a while-loop through the list and use local variables to temporally save the current largest element. We use Java and JML [LBR98] as programming and specification language in the example.

In Listing 1, we start implementing method `maxElement` that is specified with a pre- and postcondition contract. The precondition states that the list must contain at least one element. The postcondition states that the largest element in the list is returned. In this example, we start with a program where some CbC refinement rules are already applied, and then, apply the block rules to finish the implementation.

The program is already split into three parts using the composition refinement rule with two intermediate conditions between them. In the first part, two local variables `i` and `j` are introduced with the assignment rule. The variable `i` is used to temporally store the largest element. In the beginning, the largest element (up to that point) is the first element in the list. The variable `j` is our loop variable to iterate through the list. In the third program part, variable `i` is returned. We start with this program state to show that `CBC-BLOCK` also supports the standard CbC approach, but we will now use the block rules to exemplify the benefits of `CBC-BLOCK`.

In the second part between the two intermediate conditions, the block rule of `CBC-BLOCK` is applied. The block `B1` is specified with a block contract in lines 13–16. For the functional behavior, we specify the values of `i` and `j`, and the size of the list in the precondition. This specification is equal to the preceding intermediate condition. The postcondition of the block states that `i` is the largest element. This postcondition meets the intermediate condition after the block. Therefore, we know that the program is correct under the assumption that block `B1` fulfills its specification. In the next steps, we concretize the block, and the applied refinement rule guarantees that the instantiated block is correct according to its specification. We can concretize the block either in one step, by instantiating the block with concrete Java code or stepwise by instantiating the block with some Java statements and other blocks.

We decide to partially implement the block. In Listing 2, we define the block that should be refined by referring to block `B1` in line 1 and repeat the specification of that block. Inside the curly brackets the instantiation is shown. We implement the block with a while-loop. We iterate through the list as long as variable `j` is smaller than the size of the list. This is stated in the loop guard. The loop is specified with a loop invariant in lines 10–12. Thereby,

<sup>3</sup><https://github.com/KIT-TVA/CorC>



```

1  /*@ public normal_behavior
2     @ requires list.size() > 0;
3     @ ensures list.contains(\result) &&
4     @   (\forall int q; q >= 0 && q < list.size(); \result >= list.get(q));
5     @*/
6  public int maxElement(List list) {
7     int i = list.get(0);
8     int j = 1;
9
10    /*@ Intm: list.size() > 0 && i == list.get(0) && j == 1;
11
12    /*@
13       @ normal_behavior
14       @ requires list.size() > 0 && i == list.get(0) && j == 1;
15       @ ensures list.contains(i) &&
16       @   (\forall int q; q >= 0 && q < list.size(); i >= list.get(q));
17       @*/
18     { \Block B1; }
19
20    /*@ Intm: list.contains(i) &&
21    /*@   (\forall int q; q >= 0 && q < list.size(); i >= list.get(q));
22
23     return i;
24 }

```

Listing 1: Initial program of `maxElement`

the variable `i` stores the largest element of the already checked elements up to the index `j`. The index `j` is inside the bounds of the list. We use the difference between the size of the list and `j` as loop variant. As variable `j` increases in each iteration, the difference decreases, and the loop thus terminates. The increase of `j` is already implemented at the end of the while-loop. The body of the loop contains another block `B2` in lines 15–22. The precondition of the block is the loop invariant with the difference that we know that variable `j` is smaller than the size of the list. This block should update variable `i` that contains the largest element. We want to compare the largest element with the next element in the list. If that element is larger, variable `i` is updated. We checked one more element of the list, and therefore, we increase the range of the universal quantifier in the postcondition. This instantiation condenses the application of three CbC refinement rules, the repetition rule to create the loop, a composition rule, and an assignment rule for the loop body.

The next step is to verify that the instantiation satisfies the block contract. Starting with the precondition and after executing the introduced instantiation, the postcondition of the block contract must be fulfilled. The details of checking this instantiation are explained in the next section. When the correctness of this instantiation is shown, we can continue to instantiate the next block `B2`.

In Listing 3, the instantiation of block `B2` implements the case when a larger element is found. The functional pre- and postcondition of the block differ by the range of considered elements. In the postcondition, the range is increased by one. In this block, we compare the current largest element `i` with the element at index `j`. If the element at index `j` is larger, we

```

1  Block B1;
2
3  /*@
4   @ normal_behavior
5   @ requires list.size() > 0;
6   @ ensures list.contains(i) &&
7   @   (\forall int q; q >= 0 && q < list.size(); i >= list.get(q));
8   @*/
9   {
10  //@ loop_invariant list.contains(i) && j > 0 && j <= list.size() &&
11  //@   (\forall int q; q >= 0 && q < j; i >= list.get(q));
12  //@ decreases list.size() - j;
13     while (j < list.size()) {
14
15         /*@
16          @ normal_behavior
17          @ requires list.contains(i) && j > 0 && j < list.size() &&
18          @   (\forall int q; q >= 0 && q < j; i >= list.get(q));
19          @ ensures list.contains(i) && j > 0 && j < list.size() &&
20          @   (\forall int q; q >= 0 && q < j+1; i >= list.get(q));
21          @*/
22         { \Block B2; }
23
24         j = j + 1;
25     }
26 }

```

Listing 2: Refinement of block B1

update variable  $i$ . In the other case,  $i$  is still the largest element and not updated. Again, we condense CbC refinement rules by instantiating the block with concrete code. We have to verify that the instantiation is correct. If this is done, we have finished the refinement process because no further block or any abstract statement is left.

By guaranteeing the correctness of all refinement steps, we can conclude that the whole program is correct by construction. The resulting program is shown in Listing 4. Here, the blocks are recursively replaced with their instantiation. The specification is limited to the method contract and the loop invariant and variant annotations. By stepwise refining the program, we can detect errors when proving single refinement steps. This locality of information helps to track down errors more easily than with monolithic post-hoc verification.

**3.2. Block Refinement Rules of CbC-Block.** In this section, we describe how refinement rules are added to establish the CBC-BLOCK approach. We describe the refinement rule to introduce a block and the refinement rule to instantiate a block with concrete code.

For the block rules to be syntactically applicable, we extend Java to write a block with a name. Normally, a block in Java is just a sequence of Java statements inside curly brackets. In addition, Ahrendt et al. [ABB<sup>+</sup>16] defined block contracts to specify the behavior of a Java block similar to a method [Mey92, Lei95]. To establish a CbC refinement process, we introduce a specified block as an abstract statement in CBC-BLOCK with an according

```

1 Block B2;
2
3 /*@
4  @ normal_behavior
5  @ requires list.contains(i) && j > 0 && j < list.size() &&
6  @   (\forall int q; q >= 0 && q < j; i >= list.get(q));
7  @ ensures list.contains(i) && j > 0 && j < list.size() &&
8  @   (\forall int q; q >= 0 && q < j+1; i >= list.get(q));
9  @*/
10 {
11     if (list.get(j) > i) {
12         i = list.get(j);
13     }
14 }

```

Listing 3: Refinement of block B2

```

1 /*@ public normal_behavior
2  @ requires requires list.size() > 0;
3  @ ensures list.contains(\result) &&
4  @   (\forall int q; q >= 0 && q < list.size(); \result >= list.get(q));
5  @*/
6  public int maxElement(List list) {
7      int i = list.get(0);
8      int j = 1;
9      //@ loop_invariant list.contains(i) && j > 0 && j <= list.size() &&
10     //@   (\forall int q; q >= 0 && q < j; i >= list.get(q));
11     //@ decreases list.size() - j;
12     while (j < list.size()) {
13         if (list.get(j) > i) {
14             i = list.get(j);
15         }
16         j = j + 1;
17     }
18     return i;
19 }

```

Listing 4: Final implementation of `maxElement`

refinement rule. In the refinement rule, we use the Hoare triple notation that is also used for the classic CbC refinement rules. We focus on functional pre-/postconditions and exclude regular and irregular termination of blocks for CBC-BLOCK. For the instantiation of a block (e.g., to write a sequence of Java statements that fulfill the specification), we follow the syntax of a concrete block in Java, but we add a name for reference.

An abstract statement is refined by the block-introduction rule to a block with a name and a block contract. Thus, a block name is an abstract placeholder. The side condition of the refinement rule guarantees the correctness of the program to be developed. For the block-introduction rule, we have to check three parts. First, the precondition of the refined abstract statement must imply the precondition of the block. This ensures that the

pre-state of the block is satisfied, and the block can be executed. Second, the postcondition of the block must imply the postcondition of the refined abstract statement to continue the program after the block. Third, the block must satisfy its own contract. As the block can be seen as a Hoare triple, any CbC refinement rule can be applied to the block. The check of the side condition of the applied refinement rule guarantees the correctness of the block under development.

**Rule 1** (Block-Introduction). *Hoare triple  $\{P\}S\{Q\}$  is **refinable** to  $\{P'\} \text{Block } B \{Q'\}$  iff  $P$  implies  $P'$  and  $Q'$  implies  $Q$  and  $\{P'\} \text{Block } B \{Q'\}$  holds.*

With the block-instantiation rule, we allow to instantiate a block with concrete code that can contain further blocks (see the instantiation in Listing 2). For application, it must be checked that this instantiation fulfills the block contract. We use the capabilities of program verification. We translate the block to a method and verify whether this translated *block-method* fulfills its contract. Thus, we have to prove that the dynamic formula  $P \rightarrow \langle \text{statement}; \dots \rangle Q$  is fulfilled. Assuming the precondition, the postcondition must be satisfied after executing the statements in the block. Dynamic logic extends first-order logic with two operators. A diamond modality  $\langle p \rangle Q$  and a box modality  $[p]Q$  with a program  $p$  and a dynamic logic formula  $Q$ . Intuitively, the diamond modality states total correctness of the program, and partial correctness is stated with the box modality.

The translation from a block to a block-method is as follows. The block contract is translated to the contract of the block-method. The translated block-method is added to the same class as the method in which the block is declared. The statements within the block become the body of the block-method. As block could introduce local variables that are already declared in the surrounding method [KFFD86], an  $\alpha$ -conversion [Bar84] is necessary to safely rename identifiers. A block does not have the same scope of a complete method and neither has parameters nor a return type. Declarations of parameters and local variables have to be added to the block-method, so that it has the same scope as the method. Therefore, we translate accessible variables of the block to parameters of the block-method, and assignable variables of the block to fields of the class containing the block-method. This differentiation is done because a contract can only access parameter values before execution of the method, but it can access the modified values of fields. Accessible or assignable fields of the class are usable because the block-method is added to the class for verification purposes. The return type of the block-method is `void` because we exclude the use of return statements inside the block. This transformation is limited in its expressiveness as we are excluding irregular termination, but sufficient to demonstrate the correctness-by-construction process for normal execution.

**Rule 2** (Block-Instantiation). *Hoare triple  $\{P\} \text{Block } B \{Q\}$  is **refinable** to  $\{P\} \langle \text{statement}; \dots \rangle \{Q\}$  iff  $P \rightarrow \langle \text{statement}; \dots \rangle Q$ , where  $\langle \text{statement}; \dots \rangle$  is any sequence of concrete program statements possibly containing further blocks.*

**3.3. Discussion.** In this subsection, we discuss the block refinement rules in comparison to related approaches that allow to introduce code sequences, such as method calls, macro expansions, and abstract execution.

**Difference to the Method Call Rule.** The difference between the block refinement rules and the method call refinement rule is that for a method call only the contract is used to verify correctness of the caller. The content of the method is assumed to be correct with respect to the method’s contract. With the block rules, both the contract and the content of the block are always checked for correctness. A big difference between the block rules and the method call rule is their scope. In a method, only variables of the method are changed and no local variables of the calling context. A block allows the modification of local variables as demonstrated in the motivating example.

**Difference to Macro Expansion.** Macro expansion is a textual transformation of input source code. A preprocessor replaces macros with concrete source code. This is similar to the block-instantiation rule, where a block name is replaced with concrete source code. As for our block-instantiation rule, a macro expansion can capture identifiers already used in the surrounding scope. Therefore, hygienic macro expansion uses  $\alpha$ -conversion [Bar84] to rename identifiers. The difference to CBC-BLOCK is that CBC-BLOCK demands a specification for a block that is introduced in the block-introduction rule. Additionally, the block-instantiation rule starts a procedure that verifies whether the block instantiation fulfills its specification. A macro expansion is just a transformation of code.

**Abstract Execution for Correctness-by-Construction.** Abstract execution [SH19] is a technique to specify and verify programs with partially abstract parts. Abstract execution generalizes symbolic execution. It is tailored to Java, but the principles are applicable to other sequential languages. Java and JML are extended with the concept of abstract program element (APE); an abstract statement or an abstract expression. An APE is a placeholder for any program part with or without side effects. To verify the correctness of programs containing abstract program elements, these elements are specified with a contract similar to a block contract. The extended specification language of abstract execution allows to specify the behavior of the program element in cases of regular or irregular termination including side effects [SH19]. The strength of abstract execution is the reasoning of irregular termination that we exclude in CBC-BLOCK.

APEs can be used similar to blocks of CBC-BLOCK to establish a process for refinement-based program construction. With abstract execution, we write programs containing APEs. These programs can be verified to be correct under the assumption that the APEs fulfill their specifications. In a refinement step, an APE is replaced by a program part that contains concrete statements and possibly other abstract program elements. We have to verify that the insertion fulfills the specification of the refined APE. This refinement is repeated until no APE remains. Similar to classic CbC, this process does not require a program to be monolithically verified, but it is sufficient that each APE replacement is verified to conclude that the program is correct by construction. This process is the same as for CBC-BLOCK if we always instantiate a block without using any other CbC refinement rule. We still argue that the application of other CbC refinement in tandem with blocks is beneficial because they enforce a structured program construction process where developers think about the implementation more thoroughly. Therefore, we decided for CBC-BLOCK as presented instead of utilizing abstract execution because CBC-BLOCK is the sweet spot between expressiveness and changes to the program construction process of classic CbC. Combining classic CbC refinement rules with abstract execution requires major changes to

classic CbC, so that the strength of abstract execution is usable (i.e., the refinement rules must be adapted to consider irregular termination).

**3.4. Implementation.** In this subsection, we describe the implemented tool support for CBC-BLOCK. Classic CbC is already supported by the CORC tool [RSC<sup>+</sup>19]. CORC has a graphical and a textual editor to develop programs. In this work, we extend CORC with the new rules of CBC-BLOCK. The textual IDE is implemented in Eclipse with Xtext.<sup>4</sup> Xtext provides the functionality to develop IDEs for domain-specific languages. We use Xtext to establish an editor for CORC-programs that consist of JML, Java, and the CbC-specific keywords. The grammar of a CORC-program, which represents a refinement-based construction according to CbC, is defined in Xtext. Based on this grammar, Xtext supports syntax checks, highlighting and auto completion.

In CORC, we implemented the refinement rules of Definition 1. For CBC-BLOCK, we added refinement rules of block-introduction and block-instantiation. An instantiation is written in a separated program starting with the name of the refined block and followed by the contract and the block’s implementation. To verify that a block-instantiation fulfills its block contract, a generator is implemented. It transforms a block-instantiation to a method and starts the verification process by calling KeY [ABB<sup>+</sup>16]. The transformation to a method follows the concept presented for the block-instantiation rule before. The generator also creates the final method implementation if all refinement steps are proven. All block instantiations must be recursively inserted to get the final method implementation. The final method implementation can be integrated into an existing code base. This generator can also construct partial methods when some parts are not fully refined. This is helpful in intermediate steps of the construction to retain an overview of the current method.

**3.5. Evaluation with a User Study.** We evaluate CBC-BLOCK with a user study. We compare CBC-BLOCK with classic CbC by (dis)allowing the use of the block rules to answer the following research question.

**RQ1:** Does the CBC-BLOCK approach improve classic CbC in terms of usability?

In the user study, we engaged five participants that know classic CbC and the CORC tool. Knowledge of classic CbC and CORC is a necessary prerequisite because a new feature was evaluated that could only be understood if the participants already knew the classic CbC approach in CORC. Then, they can estimate the benefits of the new block refinement rules. With this small number of participants, it was possible that everyone could solve the study tasks consecutively.

Each participant had to implement two algorithms, one algorithm with CBC-BLOCK and the block rules and one algorithm with classic CbC and without the block rules. The algorithms are `maxElement` and `dutchFlag`. The `maxElement` algorithm was already introduced as the motivating example. The `dutchFlag` algorithm sorts a list containing only three different elements. In the original description, each element has either a red, a blue or a white color. The elements of the list are to be reordered so that the list results in the national flag of the Netherlands (red, white, blue). We adapted the task to a list that contains an unknown quantity of the numbers 0, 1, and 2. Both algorithms can be implemented in a few lines of code with one loop through the list of elements. As both algorithms are explained to the participants, we expected that the correct implementation is possible without major

<sup>4</sup><https://www.eclipse.org/Xtext/>

problems. For each task, they had 30 minutes. We split the participants in two groups using the Latin Square design [WRH<sup>+</sup>12]. Each group implemented the algorithms in the same order, but the approaches were used crosswise to address possible learning effects through an order of the tools. After implementing both algorithms, we conducted a structured interview. The questions of the interview are presented in Appendix A.

We summarize the most important answers of the participants and discuss the findings. The tasks were correctly solved by all participants. In general, the participants needed more time for the task with CBC-BLOCK than for the task with classic CbC. As we only had five participants, these statistics are only of limited significance. The participants followed the CBC-BLOCK approach and refined a program stepwise using the block rule. All participants considered the introduction of the block rules to be useful. The rules save the application of other CbC refinement rules during construction. For CBC-BLOCK, the participants positively mentioned the familiarity with a textual editor, the grouping of statements for one refinement step by using a block, and the freedom to not be bound to the classic CbC refinement rules. For classic CbC, the answers are in line with previous user studies [RTC<sup>+</sup>19, RBTS21]. The participants positively mentioned the visual overview of the refinements and the status of the verification. They liked the fine-grained feedback for every applied refinement rule. As CBC-BLOCK extends CbC, these positive answers also apply for CBC-BLOCK. The participants stated for both approaches that the incremental construction helps to track down errors. The participants still miss more assistance if a proof cannot be closed.

While we observed the participants, we noticed that both approaches need a correct and sufficient specification as a starting point. If that is the case, refining and checking side-conditions can be very successful. If, on the contrary, the specification needs to be adjusted in the process, the effort to verify the program increases drastically. With classic CbC, the participants are forced into the process of refining and verifying top-down. With CBC-BLOCK, the participants have more freedom to develop the program.

Regarding the finished tasks, all participants had experience with CORC. Therefore, it is not surprising that all participants finished the task. They never used CBC-BLOCK before, but the participants conceptually understood the features of CBC-BLOCK and accepted the expansion well. Nonetheless, the participants need more time to fully understand the IDE and the programming workflow of CBC-BLOCK. This results in a longer time to implement the algorithms.

We can answer **RQ1** that CBC-BLOCK is a promising feature to increase the usability of CORC, but as for each new feature, developers need time to get used to. Some answers of the participants highlight that with more training and better tool support, they are willing to use the CBC-BLOCK approach to construct correctness-critical programs.

**Threats to Validity.** In our user study, we had only five participants. Due to the small number of participants, the qualitative results that we collected in the structured interview are not generalizable. Nevertheless, the results are relevant, since the users are experts in CORC and can therefore assess the advantages of the extension. A more comprehensive evaluation with non-experts is not possible because they cannot properly interact with the tool. The participants only implemented and verified two small algorithms in our experiment, and therefore, we cannot generalize the results to larger problems.



#### 4. TRAITCBC

In this section, we introduce TRAITCBC with a motivational example. We present TRAITCBC formally and prove soundness of the TratiCbC program construction approach. In the end of this section, we show the proof-of-concept implementation and a feasibility evaluation.

TRAITCBC uses method abstraction and method composition to enable an incremental CbC-based development approach. This approach on method level allows a flexible way to construct the desired program with any number and size of auxiliary methods. A developer starts by implementing a method (e.g., a method `a`) in a first trait. Similar to classic CbC, the method can contain holes that are refined in subsequent steps. A hole in TRAITCBC is an abstract method (e.g., an abstract method `b`) that is called in method `a`; that is, a call to an abstract method corresponds to an abstract statement in classic CbC. In the next step, one of these new abstract methods (e.g., `b`) is implemented in a second trait, again more abstract methods can be declared for the implementation. To be correct, it must be proven that each implemented method satisfies its specifications. Afterwards, the traits are composed; the composition operation checks that the specification of the concrete method `b` in the second trait fulfills the specification of the abstract method `b` in the first trait. This incremental program construction approach stops when the last abstract method is implemented, and all traits are composed.

**4.1. Motivating Example.** We illustrate using an example of how TRAITCBC enables CbC using traits. We use an object-oriented language in the code examples. In Listing 5, we construct a method `maxElement` that finds the maximum element in a list of numbers. We slightly adjust the implementation of the algorithm to better fit for TRAITCBC. With TRAITCBC, we have an abstraction on method level. We utilize methods to outsource program pieces that can be reused (i.e., we want to implement methods that are verified once, but called several times in a program to reduce verification effort).

In this `maxElement` example, a list has a head and a tail. Only non-empty lists have a maximum element. This is explicit in the precondition of our specification, where we require that the list has at least one element. In the postcondition, we specify that the result is in the list and larger than or equal to every other element. In the first step, we create a trait `MaxETrait1` that defines the abstract method `maxElement`. The method `maxElement` is abstract, i.e., equivalent to an abstract statement in CbC.

```

1 trait MaxETrait1 {
2   @Pre: list.size() > 0
3   @Post: list.contains(result) &
4     (forall Num n: list.contains(n) ==> result >= n)
5   abstract Num maxElement(List list);
6 }

```

Listing 5: Initial trait for `maxElement`

In the second step in trait `MaxETrait2` in Listing 6, we implement the method `maxElement` using two abstract methods. We introduce an `if-elseif-else`-expression where the branches invoke abstract methods. The guards check whether the list has only one element or whether the current element is larger than or equal to the maximum of the rest of the list. The

abstract method `accessHead` returns the current element, and the abstract method `maxTail` returns the maximum in the remaining list. So, we recursively search the list for the largest element by comparing the maximum element of the list tail with the current element until we reach the end of the list.

```

1  trait MaxETrait2 {
2    @Pre: list.size() > 0
3    @Post: list.contains(result) &
4      (forall Num n: list.contains(n) ==> result >= n)
5    Num maxElement(List list) =
6      if (list.size() == 1) {accessHead(list)}
7      elseif (accessHead(list) >= maxTail(list))
8        {accessHead(list)}
9      else {maxTail(list)}
10
11   @Pre: list.size() > 0
12   @Post: result == list.element()
13   abstract Num accessHead(List list);
14
15   @Pre: list.size() > 1
16   @Post: list.tail().contains(result) &
17     (forall Num n: list.tail().contains(n) ==> result >= n)
18   abstract Num maxTail(List list);
19 }

```

Listing 6: Implementation of `maxElement` with auxiliary methods

The correct implementation of the method `maxElement` can be guaranteed under the assumptions that all introduced abstract methods are correctly implemented. Similar to post-hoc verification, a program verifier conducts a proof of method `maxElement` and uses the introduced specifications of the methods `accessHead` and `maxTail`. If the proof succeeds, we know that the first method is correctly implemented. In our incremental `CbCTrait` approach, we verify each method implementation directly after construction; and so we are able to reuse each implemented method in the following steps (e.g., by calling the method in the body of other methods).

We now compose the developed traits to complete the first construction step. To perform the composition `MaxETrait1 + MaxETrait2`, we check that the specification of the method `maxElement` fulfills the specification of the abstract method in the first trait (cf. Liskov substitution principle [LW94]). In this case, this means checking that:

`MaxETrait1.maxElement(..).pre ==> MaxETrait2.maxElement(..).pre` as well as:

`MaxETrait2.maxElement(..).post ==> MaxETrait1.maxElement(..).post`.

When the composition of two verified traits is successful, the result is also a verified trait. Note that the composed trait does not need to be verified directly by a program verifier in `TRAITCBC` because it is correct by construction. In this example, the specifications are the same, thus checking for a successful composition is trivial, but this is not generally the case. In particular, the logic needs to take into account ill-founded specifications and recursion in the specification. We discuss more about the difficulties of handling those cases in previous work [RPTS22].

The methods `accessHead` and `maxTail` are implemented in the next two construction steps in traits `MaxETrait3` and `MaxETrait4`<sup>5</sup>. The implementations are shown in Listing 7 and in Listing 8. As we implement a recursive method, the method `maxTail` calls the `maxElement` method, thus `maxElement` is introduced as an abstract method in this trait. We have to verify that the method `accessHead` satisfies its specification using a program verifier. Similarly, we have to verify the correctness of the method `maxTail`.

```

1 trait MaxETrait3 {
2   @Pre: list.size() > 0
3   @Post: result == list.element()
4   Num accessHead(List list) = list.element()
5 }

```

Listing 7: Implementation of `accessHead`

```

1 trait MaxETrait4 {
2   @Pre: list.size() > 1
3   @Post: list.tail().contains(result) &
4     (forall Num n: list.tail().contains(n) ==> result >= n)
5   Num maxTail(List list) = maxElement(list.tail())
6
7   @Pre: list.size() > 0
8   @Post: list.contains(result) &
9     (forall Num n: list.contains(n) ==> result >= n)
10  abstract Num maxElement(List list);
11 }

```

Listing 8: Implementation of `maxTail`

As before, all traits are composed, and it is checked that the specifications of the concrete methods fulfill the specifications of the abstract ones. As we have no contradicting specifications for the same methods, the composition is well-formed. In Listing 9, the final program `MaxE` is shown. All traits are composed.

```

1 class MaxE = MaxETrait1 + MaxETrait2 + MaxETrait3 + MaxETrait4

```

Listing 9: Trait composition

The already proven auxiliary methods in traits can be reused. For example, if we want to implement a `minElement` method as shown in Listing 10, we could reuse already implemented traits to reduce the programming and verification effort. The method `minElement` is implemented in the following in trait `MinE` with one abstract method. The specification of the method `accessHead` is the same as for the method `accessHead` above, so `MaxETrait3` can be reused. In this example, we show the flexible granularity of `TRAITCBC` by directly implementing the else branch, instead of introducing an auxiliary method as for `maxElement`.

<sup>5</sup>The methods could also be implemented in one trait.

```

1  trait MinE {
2    @Pre: list.size() > 0
3    @Post: list.contains(result) &
4      (forall Num n: list.contains(n) ==> result <= n)
5    Num minElement(List list) =
6      if (list.size() == 1) {accessHead(list)}
7      elseif (accessHead(list) <= minElement(list.tail()))
8        {accessHead(list)}
9      else {minElement(list.tail())}
10
11   @Pre: list.size() > 0
12   @Post: result == list.element()
13   abstract Num accessHead(List list);
14 }

```

Listing 10: Implementation of minElement with auxiliary method accessHead

The correctness of `minElement` is verified with the specifications of the method `accessHead`. By composing `MinE` with `MaxETrait3`, we get a correct implementation of `minElement`. Note how this verification process supports abstraction: as long as the contracts are compatible, methods can be implemented in different styles by different developers to best meet non-functional requirements while preserving the specified observable behavior [tBCSW18]. A completely different implementation of `maxElement` can be used if it fulfills the specification of the abstract method `maxElement` in trait `MaxETrait1`. This decoupling of specification and corresponding satisfying implementations facilitates an incremental program construction approach where a specified code base is extended with suitable implementations [DDJS14].

**4.2. Object-Oriented Trait-Based Language.** In this section, we formally introduce the syntax, type system, reduction, and flattening semantics of a minimal core calculus for TRAITCBC. We keep this calculus for TRAITCBC parametric in the specification logic so that it can be used with a suitable program verifier and associated logic. The presented rules to compose traits are conventional. The focus of our work is to enable a CbC approach using traits that developers can easily adopt. Therefore, we present the calculus to prove soundness of TRAITCBC, but focus on the presentation of the advantages of incremental trait-based programming in this paper. Indeed, languages with traits and with a suitable specification language intrinsically enable incremental program construction.

**4.2.1. Syntax.** The concrete syntax of our core calculus for TRAITCBC is shown in Fig. 1, where non-terminals ending with ‘s’ are implicitly defined as a sequence of non-terminals, i.e.,  $vs ::= v_1 \dots v_n$ . We use the metavariables  $t$  for trait names,  $C$  for class names and  $m$  for method names. A program consists of trait and class definitions. Each definition has a name and a trait expression  $E$ . The trait expression can be a *Body*, a trait name, a composition of two trait expressions  $E$ , or a trait expression  $E$  where a method is made abstract, written as  $E[\text{makeAbstract } m]$ . A *Body* has a flag `interface` to define an interface, a set of implemented interfaces  $Cs$  and a list of methods  $Ms$ . Methods have a method header  $MH$  consisting of a specification  $S$ , the return type, a method name, and a list of parameters. Methods have an optional method body. In the method body, we have

$Prog$	$::=$	$Ds\ e$
$D$	$::=$	$TD \mid CD$
$Name$	$::=$	$t \mid C$
$TD$	$::=$	$t = E$
$CD$	$::=$	$C = E$
$E$	$::=$	$Body \mid t \mid E + E \mid E[\mathbf{makeAbstract}\ m]$
$Body$	$::=$	$\{\mathbf{interface?}\ [Cs]\ Ms\}$
$M$	$::=$	$MH\ e?;$
$MH$	$::=$	$S\ \mathbf{method}\ C\ m(C_1\ x_1 \dots C_n\ x_n)$
$e$	$::=$	$x \mid e.m(es) \mid \mathbf{new}\ C(es)$
$\mathcal{E}_v$	$::=$	$[\ ] .m(es) \mid v.m(vs\ [\ ]\ es) \mid \mathbf{new}\ C(vs\ [\ ]\ es)$
$v$	$::=$	$\mathbf{new}\ C(vs)$
$\Gamma$	$::=$	$x_1 : C_1 \dots x_n : C_n$
$S$	$::=$	$\dots e.g.\ \mathbf{Pre} : P\ \mathbf{Post} : P$
$P$	$::=$	$\dots e.g.\ \text{First order logic}$

Figure 1: Syntax of the trait system

standard expressions, such as variable references, method calls, and object initializations. For simplicity, we exclude updatable state. Field declarations are emulated by method declarations, and field accesses are emulated by method calls.

The specification  $S$  in each method header is used to verify that methods are correctly implemented. The specification is written in some logic. In our examples, we will use first-order logic (cf. the example in Section 4.1). A well-formed program respects the following conditions:

Every  $Name$  in  $Ds$  must be unique so that  $Ds$  can be seen as a map from names to trait expressions. Trait expressions  $E$  can refer to trait names  $t$ . A well-formed  $Ds$  does not have any circular trait definitions like  $t = t$  or  $t_1 = t_2$  and  $t_2 = t_1$ . In a  $Body$ , all names of implemented interfaces must be unique and all method names must be unique, so that  $Body$  is a map from method names to method definitions. In a method header, parameters must have unique names, and no explicit parameter can be called **this**.

**4.2.2. Typing Rules.** In our type system, we have a typing context  $\Gamma ::= x_1 : C_1 \dots x_n : C_n$  which assigns types  $C_i$  to variables  $x_i$ . We define typing rules for our three kinds of expressions:  $x$ , method calls, and object initialization. We combine typing and verification in our type checking  $\Gamma \vdash e : C \dashv P_0 \models P_1$ . This judgment can be read as: under typing context  $\Gamma$ , the expression  $e$  has type  $C$ , where under the knowledge  $P_0$  we need to prove  $P_1$ . The knowledge  $P_0$  is our collected information that we use to prove a method correct. That means, in our typing rules, we collect the knowledge about the parameters and expressions in a method body to verify that this method body fulfills the specification defined in the method header. The verification obligation  $P_1$  should follow from the knowledge  $P_0$ .

We check if methods are well-typed with judgments of form  $Ds; Name \vdash M : OK$ . This judgment can be read as: in the definition table, the method  $M$  defined under the definition  $Name$  is correct. The typing rules of Fig. 2 are explained in the following. The first four rules type different expressions and collect the information of these expressions to prove with rule MOK that a method fulfills its specification. In the rule MOK with keyword **verify**,

$$\begin{array}{c}
\frac{}{\Gamma \vdash x : \Gamma(x) \dashv \mathbf{result} : \Gamma(x) \ \& \ \mathbf{result} = x \models \mathit{true}} \quad (\text{X}) \\
\\
\frac{
\begin{array}{l}
S \ \mathbf{method} \ C \ m(C_1 \ x_1 \dots C_n \ x_n); \in \ \mathit{methods}(C_0) \quad \Gamma \vdash e_0 : C_0 \dashv P_0 \models P'_0 \dots \Gamma \vdash e_n : C_n \dashv P_n \models P'_n \\
x'_0 \dots x'_n \ \mathit{fresh} \quad S' = S[\mathbf{this} := x'_0, x_1 := x'_1, \dots, x_n := x'_n] \\
P = (\mathbf{result} : C \ \& \ P_0[\mathbf{result} := x'_0] \ \& \ \dots \ \& \ P_n[\mathbf{result} := x'_n] \ \& \ (Pre(S') \implies Post(S')))
\end{array}
}{\Gamma \vdash e_0.m(e_1 \dots e_n) : C \dashv P \models P'_0 \ \& \ \dots \ \& \ P'_n \ \& \ Pre(S')} \quad (\text{METHOD}) \\
\\
\frac{
\begin{array}{l}
\Gamma \vdash e_1 : C_1 \dashv P_1 \models P'_1 \dots \Gamma \vdash e_n : C_n \dashv P_n \models P'_n \\
\mathit{getters}(C) = S_1 \ \mathbf{method} \ C_1 \ x_1(); \dots S_n \ \mathbf{method} \ C_n \ x_n(); \quad x'_1 \dots x'_n \ \mathit{fresh} \quad S'_i = S_i[\mathbf{this} := \mathbf{result}] \\
P'_i = (P_i[\mathbf{result} := x'_i] \ \& \ (Pre(S'_i) \implies \mathbf{result}.x_i() = x'_i)) \quad P = (\mathbf{result} : C \ \& \ P'_1 \ \& \ \dots \ \& \ P'_n)
\end{array}
}{\Gamma \vdash \mathbf{new} \ C(e_1 \dots e_n) : C \dashv P \models P'_1 \ \& \ \dots \ \& \ P'_n \ \& \ Pre(S'_1) \ \& \ \dots \ \& \ Pre(S'_n)} \quad (\text{NEW}) \\
\\
\frac{\Gamma \vdash e : C' \dashv P \models P' \quad C' \ \mathit{instanceof} \ C}{\Gamma \vdash e : C \dashv P \models P'} \quad (\text{SUB}) \\
\\
\frac{
\begin{array}{l}
\Gamma = \mathbf{this} : \mathit{Name}, x_1 : C_1, \dots, x_n : C_n \quad \Gamma \vdash e : C \dashv P \models P' \\
\mathbf{verify} \ Ds \vdash (\Gamma \ \& \ Pre(S) \ \& \ P) \models (P' \ \& \ Post(S))
\end{array}
}{Ds; \ \mathit{Name} \vdash S \ \mathbf{method} \ C \ m(C_1 \ x_1 \dots C_n \ x_n) \ e; : OK} \quad (\text{MOK}) \\
\\
\frac{}{Ds; \ \mathit{Name} \vdash S \ \mathbf{method} \ C \ m(C_1 \ x_1 \dots C_n \ x_n); : OK} \quad (\text{ABSMOK}) \\
\\
\frac{
\begin{array}{l}
\mathit{Body} = \{\mathbf{interface?} \ [Cs] \ M_1 \dots M_n\} \\
Ds; \ \mathit{Name} \vdash M_1 : OK \dots Ds; \ \mathit{Name} \vdash M_n : OK
\end{array}
}{Ds; \ \mathit{Name} \vdash \mathit{Body} : OK} \quad (\text{BODYTYPED})
\end{array}$$

Figure 2: Expression typing rules of TRAITCBC

we call a verifier to prove each method once. Abstract methods (ABSOK) are always correct. Rule BODYTYPED ensures that all methods in a body are correctly typed.

**x** : As usual, the type of a variable is stored in the environment  $\Gamma$ . From the verification perspective, we do not need to prove anything to be allowed to use a variable; thus we use *true*. We know that the result of evaluating a variable is the value of such variable, and that such value is of the type of the variable; thus we have  $\mathbf{result} : \Gamma(x) \ \& \ \mathbf{result} = x$ . The **result** is the returned value of evaluating this expression, and *variable : type* is a predicate in our system. As you can notice, we are assuming that our parametric logic supports at least a logical *and* ( $\&$ ); but of course other ways to merge knowledge could work too.

**Method:** As usual, to type a method call, we inductively type the receiver and all the parameters. In this way, we obtain all the types  $C_0 \dots C_n$ , all the knowledge  $P_0 \dots P_n$ , and all the verification obligations  $P'_0 \dots P'_n$ . Inside of all conditions  $P_i \models P'_i$  we call the result of  $e_i$  **result**. We cannot simply merge the knowledge of  $P_0 \dots P_n$ , since their **result** refers to different concepts. Thus, we chose fresh  $x'_0 \dots x'_n$  variables, and we rename **result** of  $P_i$  and  $P'_i$  into  $x'_i$ . Similarly,  $S'$  is the specification of the method adapted using  $x'_0 \dots x'_n$ .

The verification obligation of course contains all the obligations of the receiver and the parameters, but also requires the precondition of the method to hold.

The knowledge contains the knowledge of the receiver and the parameters, and the method specification in implication form. Naively, one could expect that since the precondition is already in the obligation we could simply add the postcondition to the

knowledge. This would be unsound. By using the specification in implication form, the system prevents circular reasoning: we could otherwise use the postcondition to prove the precondition. Instead, when the system shows that the precondition of  $S'$  holds, it can assume the postcondition of  $S'$ . Similar to logical *and* above, we are assuming that our parametric logic supports at least logical *implication*, but of course other forms of logical consequence could work too.

Note that the postcondition will contain information about the result of the method body as information on the `result` variable.

**New:** As usual, to type an object instantiation, we inductively type all the parameters. In this way we obtain all the types  $C_1 \dots C_n$ , all the knowledge  $P_1 \dots P_n$ , and all the verification obligations  $P'_1 \dots P'_n$ . As we did for METHOD we use fresh variables to be able to compose predicates.

As we mentioned above, we rely on abstract state operations to represent state: that is, all the abstract methods in  $C$  need to be of form  $S_i$  `method`  $C_i$   $x_i()$ ; where `this.xi()` returns the value of field  $x_i$ , that in turn was initialized with the result of expression  $e_i$ . The function  $getters(C)$  returns all methods of this form.

Knowledge  $P'_i$  contains the knowledge of  $P_i$  (from expression  $e_i$ ) and it links such knowledge to the result of calling method `result.xi()`, so that calling a getter on the created object will return the expected value. However, the information is conditional over verifying the precondition of such getter. Note that we do not need to add the knowledge of the postcondition of  $x_i()$  here; this will be handled by the METHOD rule when  $x_i()$  is called.

Knowledge  $P$  is simply merging the accumulated knowledge; while the final obligation in addition to merging the accumulated obligations also requires that the precondition of all the getters hold. In this way the getter preconditions behave like the precondition of the constructor. By requiring those preconditions, we ensure that we can call the getters on all the created objects.

**Sub:** The subsumption rule is standard. We allow subtyping between class names. Note that we do not apply weakening and strengthening of conditions here.

Besides of typing correct programs, the typing rules of Trait-CbC have the goal to verify the correctness of method implementations. The following rules check whether a method or a *Body* are correct. The check for a correct method declaration in MOK calls a program verifier to verify the correctness. We need just one verifier call for the verification of each method because the rules above collected all needed knowledge and obligations.

**MOK:** In MOK, we construct a  $\Gamma$ , and we type the method body, obtaining knowledge  $P$  and obligation  $P'$ . The program verifier will know the type information of  $\Gamma$ , the premise of the method, and the knowledge  $P$ , and will prove the obligation  $P'$  and the postcondition of the method. This verification in the typing rule is indicated by the keyword **verify**. Here, we use implication, but a different program verifier may use a different form of logical consequence. The program verifier can access the specification of all the other methods since we also provide the declaration table.

**AbsMOK:** Abstract methods are correctly typed.

**BodyTyped:** A *Body* is correctly typed, if all the methods in the declaration of the *Body* are correctly typed.



$$\begin{array}{c}
\frac{Ds \vdash e \rightarrow e'}{Ds \vdash \mathcal{E}_v[e] \rightarrow \mathcal{E}_v[e']} \quad (Ctx) \\
\\
\frac{S \text{ method } C \ m(C_1 \ x_1, \dots, C_n \ x_n) \ e; \in \ \text{methods}(C)}{Ds \vdash \text{new } C(vs).m(v_1 \dots v_n) \rightarrow e[\text{this} = \text{new } C(vs), \ x_1 = v_1, \dots, \ x_n = v_n]} \quad (\text{MCALL}) \\
\\
\frac{\text{abs}(Ds(C)) = S_1 \text{ method } C_1 \ x_1(); \dots \ S_n \text{ method } C_n \ x_n();}{Ds \vdash \text{new } C(v_1 \dots v_n).x_i() \rightarrow v_i} \quad (\text{GETTER})
\end{array}$$

Figure 3: Reduction rules of TRAITCBC

4.2.3. *Reduction Rules.* We formulate three reduction rules for our system to evaluate input expressions to final values. We introduce an evaluation context  $\mathcal{E}_v$  in our syntax in Fig. 1 to define the order of evaluation. The rules of Fig. 3 are explained in the following.

**Ctx:** This is the conventional contextual rule, allowing the execution of subexpressions.

**Mcall:** We reduce a method call to an expression  $e$ , where the receiver is replaced with  $\text{new } C(vs)$ , and each parameter  $x_i$  with the actual value  $v_i$ . We also ensure that the method is declared in the class  $C$ .

**Getter:** In our formalism, abstract methods without arguments represents getters. Notation  $\text{abs}(Body)$  returns the set of all abstract methods in  $Body$ . A valid class can only have abstract methods without arguments, and they will all represent getters.

4.2.4. *Flattening Semantics.* When we implement methods in several traits, we have to check that these traits are compatible when they are composed. This process to derive a complete class from a set of traits is called flattening. We follow the traditional flattening semantics [DNS<sup>+</sup>06]. A class that is defined by composing several traits is obtained by flattening rules. All methods are direct members of the class [DNS<sup>+</sup>06]. Overall, our flattening process works as a big step reduction arrow, where we reduce a trait expression into a well-typed and verified body.

To introduce our flattening rules in Fig 4, we first define the helper functions. The function *allMeth* collects all method headers with the same name as  $m$  in all input bodies (Definition 1). When two *Bodys* are composed (Definition 2), the implemented interfaces are united and the methods are composed. The composition of methods (Definition 3) collects methods that are only defined in one of the input sets. If a method is in both sets, it is composed (Definition 4). Here, we distinguish four cases. If one method is abstract and the other is concrete, we have to show that the precondition of the abstract method implies the precondition of the concrete method. Additionally, the postcondition of the concrete one has to imply the postcondition of the abstract one. This is similar to Liskov's substitution principle [LW94]. The second case is the symmetric variant of the first case. In the third and fourth case, two abstract methods are composed. Here, the specification of one abstract method has to imply the specification of the other abstract method such that an implementation can still satisfy all specifications of abstract methods. If both methods are concrete, the composition is correctly left undefined. This composition error can be resolved by making one method  $m$  abstract in the *Body*, as defined in Definition 5. The resulting *Body* is similar with the difference that the implementation of the method  $m$  is omitted. The flattening rules in Fig. 4 are explained in the following in detail. In these

rules, a set of traits is flattened to a declaration containing all methods. If abstract and concrete methods with the same name are composed, Definitions 2-4 are used to guarantee correctness of the composition.

**Definition 1** (All Methods).  $allMeth(m, Bodys) =$   
 $\{MH; \mid Body \in Bodys, Body(m) = MH;\}$

**Definition 2** (Body Composition).  $Body_1 + Body_2 = Body$   
 $\{\text{interface? } [Cs_1] Ms_1\} + \{\text{interface? } [Cs_1] Ms_1\} =$   
 $\{\text{interface? } [Cs_1 \cup Cs_2] Ms_1 + Ms_2\}$

**Definition 3** (Methods Composition).  $Ms_1 + Ms_2 = Ms$

- $(M Ms_1) + Ms_2 = M (Ms_1 + Ms_2)$   
 if  $methName(M) \notin dom(Ms_2)$
- $(M_1 Ms_1) + (M_2 Ms_2) = M_1 + M_2 (Ms_1 + Ms_2)$   
 if  $methName(M_1) = methName(M_2)$
- $\emptyset + Ms = Ms$

**Definition 4** (Method Composition).  $M_1 + M_2 = M$

- $S \text{ method } C m(C_1 x_1 \dots C_n x_n) e; + S' \text{ method } C m(C_1 \dots C_n -);$   
 $= S \text{ method } C m(C_1 x_1 \dots C_n x_n) e;$   
 if  $Pre(S')$  implies  $Pre(S)$  and  $Post(S)$  implies  $Post(S')$
- $MH_1; + MH_2 e; = MH_2 e; + MH_1;$
- $S \text{ method } C m(C_1 x_1 \dots C_n x_n); + S' \text{ method } C m(C_1 \dots C_n -);$   
 $= S \text{ method } C m(C_1 x_1 \dots C_n x_n);$   
 if  $Pre(S')$  implies  $Pre(S)$  and  $Post(S)$  implies  $Post(S')$
- $S \text{ method } C m(C_1 x_1 \dots C_n x_n); + S' \text{ method } C m(C_1 \dots C_n -);$   
 $= S' \text{ method } C m(C_1 x_1 \dots C_n x_n);$   
 if  $(Pre(S)$  implies  $Pre(S')$  and  $Post(S')$  implies  $Post(S))$   
 and not  $(Pre(S')$  implies  $Pre(S)$  and  $Post(S)$  implies  $Post(S'))$

**Definition 5** (Body Abstraction).  $Body[makeAbstract m]$   
 $\{[Cs] Ms_1 S \text{ method } C m(Cxs); Ms_2\}[makeAbstract m]$   
 $= \{[Cs] Ms_1 S \text{ method } C m(Cxs); Ms_2\}$

**FlatTop:** The first rule flattens a set of declarations  $D_1 \dots D_n$  to a set  $D'_1 \dots D'_n$ . We express this rule in a non-computational way: we assume to know the resulting  $D'_1 \dots D'_n$ , and we use them as a guide to compute them. Note that if there is a resulting  $D'_1 \dots D'_n$  then it is unique; flattening is a deterministic process and  $D'_1 \dots D'_n$  are used only to type check the results. They are not used to compute the shape of the flattened code.

Non computational rules like this are common with nominal type systems [IPW01] where the type signatures of all classes and methods can be extracted before the method bodies are verified.

**DFlat:** This rule flattens an individual definition by flattening the trait expression. When the flattening produces a class definition, we also check that the body denotes an instantiable class; a class whose only abstract methods are valid getters. The function  $abs(Body)$  returns the abstract methods.

**BFlat:** It may look surprising that the *Body* does not flatten to itself. This represents what happens in most programming languages, where implementing an interface implicitly imports the abstract signature for all the methods of that interface. In the context of

$$\begin{array}{c}
\frac{D'_1 \dots D'_n \vdash D_1 \Downarrow D'_1 \quad \dots \quad D'_1 \dots D'_n \vdash D_n \Downarrow D'_n}{D_1 \dots D_n \Downarrow D'_1 \dots D'_n} \text{ (FLATTOP)} \\
\\
\frac{Ds; \text{Name} \vdash E \Downarrow \text{Body} \quad \text{if Name of form } C \text{ then } \text{abs}(\text{Body}) = S \ T \ x_1(); \dots S \ T \ x_n();}{Ds \vdash \text{Name} = E \Downarrow \text{Name} = \text{Body}} \text{ (DFLAT)} \\
\\
\frac{\text{Body} = \{\text{interface? } [Cs] \ M_1 \dots M_n\} \\ \text{Body}' = \{\text{interface? } [Cs] \ M_1 \dots M_n \ Ms\} \\ Ms = \{\Sigma \text{allMeth}(Ds, Cs, m) \mid m \in \text{dom}(Cs) \text{ and } m \notin \text{dom}(\text{Body})\} \quad Ds; \text{Name} \vdash \text{Body}' : OK}{Ds; \text{Name} \vdash \text{Body} \Downarrow \text{Body}'} \text{ (BFLAT)} \\
\\
\frac{}{Ds; \text{Name} \vdash t \Downarrow Ds(t)} \text{ (TFLAT)} \quad \frac{Ds; \text{Name} \vdash E_1 \Downarrow \text{Body}_1 \quad Ds; \text{Name} \vdash E_2 \Downarrow \text{Body}_2}{Ds; \text{Name} \vdash E_1 + E_2 \Downarrow \text{Body}_1 + \text{Body}_2} \text{ (+FLAT)} \\
\\
\frac{Ds; \text{Name} \vdash E \Downarrow \text{Body} \quad \text{Body} = \{[Cs] \ \overline{M}_1 \ S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n); \ \overline{M}_2\} \\ \text{Body}' = \{[Cs] \ \overline{M}_1 \ S \ \text{method } C \ m(C_1 \ x_1 \dots C_n \ x_n); \ \overline{M}_2\}}{Ds; \text{Name} \vdash E[\text{makeAbstract } m] \Downarrow \text{Body}'} \text{ (ABSFLAT)}
\end{array}$$

Figure 4: Flattening rules of TRAITCBC

verification also the specification of such interface methods is imported. In concrete,  $\text{Body}'$  is like  $\text{Body}$ , but we add  $Ms$  by collecting all the methods of the interfaces that are not already present in the  $\text{Body}$ .

Moreover, we check that all the methods defined in the class respect the typing and the specification defined in the interfaces: if a class has  $S \ \text{method } \text{Foo } \text{foo}();$  or  $S \ \text{method } \text{Foo } \text{foo}() \ \mathbf{e};$  and there is a  $S' \ \text{method } \text{Foo } \text{foo}();$  in the interface, then  $S$  must respect the specification  $S'$ . The system then checks that the  $\text{Body}$  is well-typed and verified by calling  $Ds; \text{Name} \vdash M_i : OK$

**TFlat:** A trait  $t$  is flattened to its declaration  $Ds(t)$ .

**+Flat:** The composition of two expression  $E_1$  and  $E_2$ , where both expressions are first reduced to  $\text{Body}_1$  and  $\text{Body}_2$ , results in the composition of these bodies as defined in Definition 2.

**AbsFlat:** An expression  $E$  where one method  $m$  is made abstract flattens to a  $\text{Body}'$ . We know that  $E$  flattens to  $\text{Body}$ . The only difference between  $\text{Body}$  and  $\text{Body}'$  is that the one method  $m$  is abstract in  $\text{Body}'$ . In  $\text{Body}$ , the method can be abstract or concrete.

4.2.5. *Soundness of TRAITCBC.* In this subsection, we formulate the main result of the TRAITCBC approach. We prove soundness of the flattening process with a parametric logic. We claim that if you have a language without code reuse and with sound and modular post-hoc verification then the language supports CbC simply by adding traits to the language. That is, traits intrinsically enable a CbC program construction approach.

To prove soundness of the construction approach of TRAITCBC (Theorem 2: Sound CbC Process) as exemplified in Section 4.1, we have to show that the flattening process is correct (Theorem 1: General Soundness). In turn, to prove General Soundness, we need two lemmas which state that the composition of traits is correct (Lemma 1) and that a trait after the `makeAbstract` operation is still correct (Lemma 2).

In Lemma 1, we have well-typed definitions  $Ds$ , and two well-typed and verified traits in  $Ds$ , and the resulting trait/class is also well-typed and verified.

**Lemma 1** (Composition correct).

If  $Ds(t1) = Body_1$ ,  $Ds(t2) = Body_2$ ,  $Ds(Name) = Body$ ,  $Ds; t1 \vdash Body_1 : OK$ ,  $Ds; t2 \vdash Body_2 : OK$ , and  $Body_1 + Body_2 = Body$ ,  
then  $Ds; Name \vdash Body : OK$

*Proof.* We prove by contradiction. We assume the resulting  $Body$  is ill typed. By definition of BODYTYPED, it means that one of the methods cannot be typed with either ABSMOK or MOK. The list of methods that need to be typed is obtained by Definition 2.

Abstract methods can only be typed with ABSMOK and are never wrong. Implemented methods can only be typed with MOK. If  $\Gamma \vdash e : C \dashv P \models P'$  or the other precondition **verify**  $Ds \vdash (\Gamma \& Pre(S) \& P) \models (P' \& Post(S))$  does not hold, it means that there was a method  $m_i$  with expression  $e_i$  in  $Body_1$  (or symmetrically for  $Body_2$ ) that was well-typed under  $Ds; t1 \vdash Body_1$ . That means that all of its implemented methods were well-typed and verified. Typing  $e_i$  produces  $P_i \models P'_i$  by using a  $\Gamma_{t1}$  containing **this** :  $t1$ .

If  $Ds; Name \vdash Body : OK$  is not applicable, the same expression  $e_i$  was typed using a  $\Gamma_{Name}$  containing **this** :  $Name$ . It produced  $P''_i \models P'''_i$  so that **verify**  $Ds \vdash (\Gamma_{Name} \& Pre(S) \& P''_i) \implies (P'''_i \& Post(S))$  does not hold. We know that **verify**  $Ds \vdash (\Gamma_{t1} \& Pre(S) \& P_i) \implies (P'_i \& Post(S))$  holds by our assumption. By Definition 4, the contracts of the methods in  $Body$  are simply stronger than the contracts of the methods in  $Body_1$ . The only difference between  $P''_i \models P'''_i$  and  $P_i \models P'_i$  is in the contracts of methods called on **this**. Assuming that our parametric logic implication is transitive, we know that **verify**  $Ds \vdash (\Gamma_{t1} \& Pre(S) \& P_i) \implies (P'_i \& Post(S))$  entails **verify**  $Ds \vdash (\Gamma_{Name} \& Pre(S) \& P''_i) \implies (P'''_i \& Post(S))$ , thus we reach a contradiction.  $\square$

Lemma 2 shows that if we have a well-typed and verified trait, the operation **makeAbstract** results in a trait/class that is also well-typed and verified.

**Lemma 2** (MakeAbstract correct).

If  $Ds(t) = Body$ ,  $Ds(Name) = Body'$ ,  $Ds; t \vdash Body : OK$ ,  
and  $Body[\mathbf{makeAbstract} \ m] = Body'$ ,  
then  $Ds; Name \vdash Body' : OK$

*Proof.* We prove by contradiction. We assume the resulting  $Body'$  is ill typed. By definition of BODYTYPED, it means that one of the methods cannot be typed with either ABSMOK or MOK. The list of methods that need to be typed is obtained by Definition 2.

Abstract methods can only be typed with ABSMOK and are never wrong. We know that  $Body$  is typable by our assumption. The only difference between  $Body$  and  $Body'$  is that the method  $m$  is made abstract. As we have seen for Lemma 1, we are typing  $Body'$  in a different  $\Gamma$ . This case is even simpler than Lemma 1 because  $Body$  and  $Body'$  have exactly the same specifications. The abstract method  $m$  and thus  $Body'$  cannot be ill typed.  $\square$

With these Lemmas, we can prove Theorem 1. Given a sound and modular verification language, then all programs that flatten are well-typed and verified. In a modular verification language, a method can be fully verified using only the information contained in the method declaration and the specification of any used method. Moreover, our parametric logic must support at least a commutative and associative *and* (but of course other ways to merge knowledge could work too) and a transitive *implication* (but of course other forms of logical consequence could work too).

**Theorem 1** (General Soundness).

For all programs  $Ds$  where  $Ds$  flattens to  $Ds'$ , and  $Ds'$  is well-typed; that is, for all  $Name = Body \in Ds'$ , we have  $Ds'; Name \vdash Body : OK$ .

*Proof.* By induction on the size of  $Ds$ , and by induction on cases of  $E$  (the applied flattening rule for  $E$ ).

- $Body$  only flattens if the  $Body$  can be shown to be well-typed.
- $t$  only reads a trait from the already verified  $Ds'$ .
- $Body_1 + Body_2$  is correct with Lemma 1. The lemma can be applied directly, if  $E$  is of depth one (e.g.,  $Body_1 + Body_2$ ). If  $E$  is more complex, we have to apply other cases of this case analysis.
- `makeAbstract` is handled similarly using Lemma 2.
- By the flattening relation, we know that  $Body_1$  and  $Body_2$  are well-typed in  $Ds$ . If we start from a program containing only well-typed and verified traits, any new class we can define by just composing those traits is well typed and verified.

□

We now show that the TRAITCBC approach is sound. Theorem 2 states that starting with one abstract method and a set of verified traits, the composed program is also verified.

**Theorem 2** (Sound CbC Process).

Starting from a fully abstract specification  $t_0$ , and some construction steps  $t_1 \dots t_n$ , we can write  $C = t_0 + \dots + t_n$  as our whole CbC approach, where  $t_0 + t_1$  is the application of the first construction step. If we use CbC to construct programs, we can start from verified atomic units and get a verified result. Formally, if  $t_0 = \{MH\}$   $t_1 = \{Ms_1\}$  ...  $t_n = \{Ms_n\}$  are well-typed, and

$$\begin{array}{lcl} t_0 = \{MH\} & & t_0 = \{MH\} \\ t_1 = \{Ms_1\} \dots t_n = \{Ms_n\} & \Downarrow & t_1 = \{Ms_1\} \dots t_n = \{Ms_n\} \\ C = t_0 + \dots + t_n & & C = Body \end{array}$$

then  $C = Body$  is well-typed.

*Proof.* This is a special case of Theorem 1. □

Theorem 2 shows clearly that trait composition intrinsically enables a CbC approach: An object-oriented programming language with traits and a corresponding specification language supports an incremental CbC approach.

**4.3. Proof-of-Concept Implementation.** In this section, we describe the implementation, which instantiates TRAITCBC in Java with JML [LBR98] as specification language and KeY [ABB<sup>+</sup>16] as verifier for Java code. Our trait implementation is based on interfaces with default implementation. Our open source tool is implemented in Java and integrated as plug-in in the Eclipse IDE.<sup>6</sup> Besides this prototype, other languages with a suitable verifier, such as Dafny [Lei10] and OpenJML [Cok11], can also be used to implement TRAITCBC.

In Listing 11, we show the concrete syntax of our implementation. Each method in a trait is specified with JML with the keywords `requires` and `ensures` for the pre- and postcondition. To verify the correctness of programs, we need two steps. First, we verify the correctness of a method implemented in a trait w.r.t. its specification. Second, for trait composition, our implementation checks the correct composition for all methods (cf. Definition 2). The

<sup>6</sup>Tool and evaluation at <https://doi.org/10.5281/zenodo.7766635>

syntax of trait composition is shown in Listing 12. In a tc-file (a file to specify the traits to be composed), the name of the resulting trait is given and the composed traits are connected with a plus operator. In Listing 12, trait `MaxElement1` is composed with trait `MaxElement2`. The trait `MaxElement2` must implement the methods `accessHead` and `maxTail`, so that we obtain a correct result in which all methods are implemented. To verify correctness of the trait composition, it is checked that the specification of a concrete method satisfies the specification of the abstract one with the same signature (cf. Definition 4). These verification goals are sent to KeY, which starts an automatic verification attempt.

```

1 public interface MaxElement1 {
2   /*@ requires list.size() > 0;
3     @ ensures (\forall int n; list.contains(n);
4     @ \result >= n) & list.contains(\result);
5   @*/
6   default public int maxElement(List list) {
7     if (list.size() == 1) return accessHead(list);
8     if (list.element() >= maxElement(list.tail()))
9       { return accessHead(list); }
10    else { return maxTail(list); } }
11
12  /*@ requires list.size() > 0;
13    @ ensures \result == list.element();
14  @*/
15  public int accessHead(List list);
16
17  /*@ requires list.size() > 1;
18    @ ensures (\forall int n; list.tail().contains(n);
19    @ \result >= n) & list.tail().contains(\result);
20  @*/
21  public int maxTail(List list);
22 }

```

Listing 11: Example of a trait in our implementation

```

1      ComposedMax = MaxElement1 + MaxElement2

```

Listing 12: Example of a trait composition

**Evaluation.** We evaluate our implementation by a feasibility study. First, we reimplemented an already verified case study in our trait-based language. We used the `IntList` [STAL11] case study, which is a small software product line (SPL) with a common code base and several features extending this code base. Here, we can show that our trait-based language also facilitates reuse. The `IntList` case study implements functionality to insert integers to a list in the base version. Extensions are the sorting of the list and different insert options (e.g., front /back). We implement five methods that exists in different variants with our trait-based CbC approach. We implement the case study in different granularities. The coarse-grained version is similar to the SPL implementation we started

	Classic CbC	CBC-BLOCK	TRAITCBC
Language	Additional refinement rules for a programming language. Needs specification language.	Additional refinement rules for a programming language. Introduces a specified block of statements. Needs specification language.	Programming language with traits. Needs specification language.
Tool support	Pen and paper. Some specialized tools available.	Pen and paper. Some specialized tools available. Block instantiation rule relies on post-hoc verification tools.	Relies on post-hoc verification tools.
Construction Rules	Specific refinement rules.	Specific refinement rules.	Construction by composition of traits.
Correctness/ Debugging	Guarantees the correctness of each refinement step.	Guarantees the correctness of each refinement step. Refinements can be condensed with the block rules.	Guarantees the correctness of each construction step. Each method is specified so that each constructed method can directly be verified.
Proof complexity	Many, but small proofs.	Any granularity of proofs.	Any granularity of proofs.
Reuse	Refinement steps cannot be reused; only fully implemented methods can.	Refinement steps cannot be reused; only fully implemented methods can.	Each verified method in a trait can be reused.
Applications	Focuses on small, but correctness-critical algorithms.	Focuses on correctness-critical algorithms.	As TRAITCBC is based on post-hoc verification, it can be used in similar areas where post-hoc verification is used. Traits are beneficial for incremental development and software product lines.

Table 1: Comparison of TRAITCBC with CBC-BLOCK and classic CbC

with [STAL11], confirming that traits are also amenable to implement SPLs as shown by Bettini et al. [BDS10]. The fine-grained version implements the five methods incrementally with 12 construction steps. We can reuse 6 of these steps during the construction of method variants.

We also implement three more case studies (BankAccount [TSAH12], Email [Hal05], and Elevator [PR01]) with TRAITCBC and classic CbC to show that it is feasible to implement object-oriented programs with both approaches. We used CORC [RSC<sup>+</sup>19] as an instance of a classic CbC tool. We were able to implement 9 classes and verify 34 methods with a size of 1–20 lines of code. For future work, a user study is necessary to evaluate the usability of TRAITCBC in comparison to classic CbC to empirically confirm our stated advantages.

## 5. THE DIFFERENT CBC-BASED PROGRAM CONSTRUCTION APPROACHES IN COMPARISON

In this section, we discuss classic CbC in comparison to CBC-BLOCK and TRAITCBC. In Table 1, we summarize how the three approaches compare regarding main aspects of developing correct programs using tool support. The aspects comprise the programming language, the tool support, the procedure to develop programs, and the verification of the program.

**Language.** All approaches need an underlying programming and specification language. The defined refinement rules of the classic CbC approach are external to a programming



language. That means, each refinement rule introduces some statement of the programming language by transforming the program. With CBC-BLOCK and the block-instantiation rule more than one statement of the language can be introduced at once. TRAITCBC is usable with languages that have traits. Methods can be implemented as defined by the language. No refinement rules are necessary.

**Tool Support.** Tool support is helpful for any of the approaches. For classic CbC, mostly pen and paper is used. There are a few specialized tools such as CORC [RSC<sup>+</sup>19], ArcAngel [OCW03], and SOCOS [Bac09, BEM07]. These tools force a certain programming procedure on the user because refinement rules must be applied to implement programs. CBC-BLOCK is implemented in CORC and extends the set of refinement rules with the new rules for blocks. To verify the correctness of block instantiations, program verifiers can be reused. There are program verifiers for many languages, such as Java [ABB<sup>+</sup>16], C [CDH<sup>+</sup>09], and C# [BFL<sup>+</sup>11, BLS04]. Other languages are integrated with their verifier from the start, e.g., Spec# [BLS04] and Dafny [Lei10]. For TRAITCBC, we also need a program verifier to prove the correctness of method implementations, but we do not need specialized tools to construct methods, such as CORC.

**Construction Rules.** To construct a program, classic CbC has a strict concept of refinement rules that must be applied to construct a program. CBC-BLOCK relaxes this strict guideline to construct programs. Programs can be constructed stepwise as with classic CbC, but if desired, any number of refinement steps can be condensed with the block rules. In the extreme case, a whole program can be developed in one step. TRAITCBC offers this flexibility to construct programs without the need of external refinement rules. Methods can be developed freely and only need to be composed with respect to their specification. Nevertheless, TRAITCBC supports to construct code in fine-grained steps, which are more amenable for verification than more complex methods.

**Correctness/Debugging.** Classical CbC gives explicit information about the program states before and after execution of each statement by the Hoare triple notation. The correctness of each applied refinement step is guaranteed by proving the side conditions of the refinement rule. Some side conditions are not directly provable because abstract statements in Hoare triples must be concretized first. In the worst case, a problem in the program is found only after some refinement steps. The abstract statements in classic CbC are not explicitly specified by the developer. Additional specifications in classic CbC are introduced with some rules such as an intermediate condition in the composition rule. Then, these specifications are propagated through the program to be constructed. Again, due to a possible delayed check of a side condition, a wrong specification is found only after some refinement steps.

If errors occur in the program development process, TRAITCBC gives early and detailed information on the level of verified methods. By specifying the method under development and any abstract method that is called by this method, we can directly verify the correctness of the method under development. We assume that the introduced abstract methods will be correctly implemented in further refinement steps. With each step, the developer gets closer to the solution until finally all abstract methods are implemented. CBC-BLOCK combines the characteristics of the other two approaches. The refinement rules of classic CbC can be applied, or blocks of statements can be introduced. The specified block is verified similar to a method in TRAITCBC.

**Proof Complexity.** Classical CbC requires many small proofs to guarantee the correctness of a program. CBC-BLOCK can condense the proofs into larger proofs using the block

refinement rules. TRAITCBC can have the same granularity and also the same proof effort as classic CbC, since each method implementation can correspond to just one refinement step. The advantage of TRAITCBC and CBC-BLOCK is that developers can freely implement a method body or a block. They must not stick to the same granularity as in the classic CbC refinement rules. Proof complexity can be balanced against verifier calls.

**Reuse.** A fully refined method can be reused in all approaches. For TRAITCBC, we can easily reuse even very small units of code, since they are represented as methods in the traits. In classic CbC and CBC-BLOCK, no refinement step can be reused.

**Applications.** The classic CbC approach does not scale well to development procedures for complete software system. Rather, individual algorithms can be developed with CbC [WKSC16]. With the block rules, the scalability is improved because refinement steps that are easy to prove can be combined into one block. This saves the application of refinement rules and their corresponding correctness proofs. With ARCHICORC [KRS20], we can even scale CbC to the development of correct component-based architectures. By composing components specified with required and provided interfaces, we support the creation of software architectures correct by construction.

As soon as we scale TRAITCBC to real languages, we have the same application scenarios as approaches that already use post-hoc program verification. As argued by Damiani et al. [DDJS14], traits enable an incremental process of specifying and verifying software. Bettini et al. [BDS10] proposed to use traits for software product line development and highlighted the benefits of fine-grained reuse mechanisms. Here, TRAITCBC’s guideline is suitable for constructing new product lines step by step from the beginning.

Since CBC-BLOCK extends classic CbC and can be freely applied at any granularity of refinement steps, we propose to use CBC-BLOCK for any implementation of correctness-critical software, but the CbC approach must be well understood by the developer to be efficiently usable. In TRAITCBC, methods are developed and composed directly, so less knowledge is needed to apply the approach, but developers can fall back into a post-hoc verification process and thus lose the benefits of CbC (e.g., if the developers first develop all methods and do not directly prove the correctness). In general, both approaches are usable for program development and the right choice depends on the preferences and prior knowledge of the developers.

**Summary.** In summary, TRAITCBC and CBC-BLOCK allow more flexible program construction without losing the advantages of incremental correct-by-construction program development. CBC-BLOCK loosens the strict guideline of classic CbC by adding the block refinement rules. CBC-BLOCK still needs specialized tools, such as CORC to be applicable. TRAITCBC enables a CbC approach for trait-based languages without introducing refinement rules. This program construction approach combined with the flexibility of traits allows correct methods to be developed in small and reusable steps. TRAITCBC is independent of special CbC tools and requires only a program verifier.

## 6. RELATED WORK

In the following, we discuss related work for specifying and verifying software. We discuss related correctness-by-construction approaches and compare CORC with other tools for CbC.

**Contracts and Program Verification.** The implementation of CbC in CORC and the implementation of TRAITCBC use JML, Java DL and Java to specify and write programs. For the verification, KeY [ABB<sup>+</sup>16] is integrated in the backend. KeY is a deductive program verifier for Java programs specified with JML. In an intermediate step, the specified programs are translated to Java DL. Similarly, OpenJML [Cok11] verifies Java programs specified with JML. Besides Java/JML, many languages support pre-/postcondition contracts or other forms of specification to state program behavior. First, the programming language Eiffel introduced contracts and supported the design-by-contract approach [Mey88, Mey92]. Eiffel is an object-oriented programming language, where classes are specified with invariants, and methods with pre-/postconditions contracts. For the verification, AutoProof [KRMJ16, TFNP15] is integrated that translates the specified program to a logic formula. Then, an SMT-solver proves the validity of the formulas. For C#, the language Spec# is an extension to introduce contracts and invariants [BLS04, BFL<sup>+</sup>11]. The verification is done by translating the proof obligations to an intermediate language BoogiePL that can be verified with Boogie [BCD<sup>+</sup>05]. For the C language, the VCC [CDH<sup>+</sup>09] and Frama-C [CKK<sup>+</sup>12] tools verify annotated C code. VCC reuses the Spec# tool chain. For Java and C, the VeriFast [JSP10] tool verifies C and Java programs. VerCors [ABD<sup>+</sup>14] also support the verification of C and Java programs with a focus on concurrent and distributed software. Another language with integrated specifications and verification is Dafny [Lei10]. Dafny is a functional language, but supports the compilation to other languages such as C#, Java, Go, and Python. Similarly, Whiley [PG13] is a designed language with associated verifier to simplify the verification of programs. The languages SPARK [Bar03] supports a subset of the Ada language to specify and verify Ada programs. In contrast to JML, the specification is not written as comments, but the Ada *aspect*-syntax is used to express contracts. The focus of all these languages and verification tools is the specification of program behavior and the verification that a program satisfies its specification. With CbC (CBC-BLOCK and TRAITCBC), we put the correct construction of programs in the foreground, instead of just verifying the correctness post-hoc. However, Watson et al. [WKSC16] argue that correctness-by-construction and post-hoc verification can be used together to combine their mutual strengths.

To verify trait languages, Damiani et al. [DDJS14] added specifications of methods in traits to verify correct trait composition. They proposed a modular and incremental verification process. Traits are introduced in many languages to support clean design and reuse, for example Smalltalk [DNS<sup>+</sup>06], Java [BMN14] by utilizing default methods in interfaces, and other Java-like languages [BDSS13, LS08, SD05]. None of these trait languages were used to formulate a CbC approach to create correct programs. They only focus on code reuse or post-hoc verification.

**Refinement-Based Correctness-by-Construction.** The main idea of correctness-by-construction is the stepwise construction of a program from a starting specification with correctness guarantees for each step. We focused on correctness-by-construction by Kourie and Watson [KW12] that we called classic CbC. This classic CbC approach is based on Dijkstra [Dij76] and Gries [Gri87]. In this paragraph, we discuss related refinement-based CbC approaches. All of these approaches create correct programs by refining an abstract program or system to a concrete implementation. This is the main difference to the composition-based CbC approach of TRAITCBC, where atomic units of code are composed to whole programs.

Morgan’s refinement calculus [Mor94] is similar to correctness-by-construction by Kourie and Watson [KW12]. Both approaches have the same theoretical foundation, but Morgan’s refinement calculus is more elaborated with a large number of different refinement rules, where many rules are only formally interesting. Kourie and Watson [KW12] reduced the refinement rules to a minimal but sufficient set, such that CbC becomes comprehensible for developers without a major background in formal methods. The language ArcAngel [OCW03] with the verifier ProofPower [ZOC09] implements Morgan’s refinement calculus. The tool uses a tactic language to apply a sequence of refinement rules for program refinement. Thus, a tactic has the same benefit as the application of a block refinement in CBC-BLOCK because the application of refinement rules is condensed to one refinement step. The difference is that for an introduced block of code in CBC-BLOCK, it does not matter what classic CbC refinement rules would have to be applied to introduce that block of code. A tactic still applies the refinement rules stated in that tactic sequentially.

The invariant based programming [BW12, Bac09] shifts the focus from pre-/postcondition contracts as starting point for refinements to invariants. The tool SOCOS [Bac09, BEM07] implements Back’s methodology. Similar to CORC, SOCOS has a graphical user interface to create a program in the form of a UML-style state chart. Refinement steps introduce new states and transitions in the state chart and check compliance with the invariants. A completely refined program is proved correct and executable code can be generated. In CORC, the graphical user interface present the refinement steps in a hierarchical tree structure that more directly represent the structure of the code (comparable with an abstract syntax tree). Therefore, CORC and also the implementation of TRAITCBC are on the level of source code.

Further refinement-based methodologies are Event-B [Abr10, ABH<sup>+</sup>10] for automata-based systems and Circus [OCW09, OGC08] for state-rich reactive systems. Both methodologies work on an abstraction level with abstract models instead of specified source code. In refinement steps these abstract models of the system are transformed to concrete and executable implementation. Here, each refined result guarantees conformations with the initial model. Event-B is supported by the tool Rodin [ABH<sup>+</sup>10], and Circus is supported by the tool CRefine [OGC08]. The main difference to CbC by Kourie and Watson [KW12], and TRAITCBC is the abstraction level. We specify and verify source code rather than automata-based systems.

Data refinement [HKKN13, HL22, LT12, CDM13] is a related approach that focuses on the refinement of (abstract) programs with abstract types to correct and more efficient programs with concrete types. Haftmann et al. [HKKN13] examine how the Isabelle/HOL code generator applies data refinements to produce executable versions of abstract programs. Cohen et al. [CDM13] present an approach to refine Coq programs to enhance computational efficiency. Haslbeck and Lammich [HL22] not only ensure functional correctness during data refinement, they also verify worst-case complexity of algorithms at the LLVM level. The main difference to CbC by Kourie and Watson [KW12] is that data refinement approaches start with algorithms on abstract data structures that are refined to more concrete data structures, whether CbC by Kourie and Watson focuses on the incremental development of the algorithm itself. Therefore, both approaches can be used in concert to develop more efficient algorithms.

**Extensions to Correctness-by-Construction and CorC.** CORC has been extended in several directions to allow the structured program development for larger software systems

and further application areas. With ARCHICORC [KRS20], we integrate the construction of correct software architectures. We bundle CORC programs into reusable software components. The components communicate via required and provided interfaces where ARCHICORC guarantees the compatibility between them. With VARCORC [BRS20] software product lines are developed correct by construction. A software product line is used to systematically construct a family of similar software programs instead of developing monolithic programs. VARCORC ensures the correctness of all possible software variants of the product line. In addition to functional correctness, correctness-by-construction and CORC are extended to guarantee nonfunctional properties. As a first example, we introduced CbC refinement rules to ensure that programs [RKTS20, RKS<sup>+</sup>22] follow an information flow policy which defines the allowed flow of information in a program. In every refinement step, security and functional correctness of the program is guaranteed, such that insecure and incorrect programs are prohibited by construction. The goal of these extensions is that program development in CORC is scalable and that CbC can be used for additional application areas. Orthogonally, this article focuses on improving the flexibility of developing programs correct by construction (e.g., by introducing the block refinement rules).

**Program and Specification Synthesis.** Program synthesis is a technique that generates programs from user given specifications automatically. Pioneers in this field are Manna et al. [MW80]. Gulwani et al. [GPS<sup>+</sup>17] give an overview of state-of-the-art program synthesis approaches. For example, for Fortran, Stickel et al. [SWL<sup>+</sup>94] deductively extract programs from user-given graphical specifications. They compose procedures from libraries to full implementations. Similarly, Gulwani et al. [GJTV10] synthesize programs by composing base components from a specified library. Polikarpova et al. [PKSL16] synthesize recursive programs from specifications by utilizing type information. Similarly, synthesis of function summaries [Hoa71, CDK<sup>+</sup>15, SFS12] automatically generate pre-/postcondition specifications from programs to achieve modular verification and to improve verification time. With CbC (classic CbC, CBC-BLOCK, or TRAITCBC), developers have the task to specify and create programs according to that specification. Therefore, CbC is a program development approach where the developer determines the resulting program, while program synthesis generates one of possibly many programs that fulfill the specification. Contrary to this, the synthesis of a function summary generates one of possibly many specifications for a program. Synthesis has scalability limitations due to an enormous search space of programs/specifications and ambiguity of user intent.

## 7. CONCLUSION

In this article, we presented CBC-BLOCK and TRAITCBC two incremental program construction approaches that guide developers to implementations that are correct by construction. CBC-BLOCK extends classic CbC with block refinement rules. These rules allow to condense the application of CbC refinement rules into one block refinement. Thus, CBC-BLOCK increase flexibility in the development of programs because any sequence of statements can be introduced in a block, while still ensuring the correctness of that introduced block. TRAITCBC uses method calls and trait composition instead of refinement rules to guarantee functional correctness. We formalize the concept of a trait-based object-oriented language with a parametric specification language to allow a broader range of languages to adopt this concept. The main advantage of TRAITCBC is the simplicity of the refinement process that

supports code reuse. We compared classic CbC, CBC-BLOCK, and TRAITCBC qualitatively with regard to their programming constructs, tool support, and usability. CBC-BLOCK and TRAITCBC both relax the strict guideline of CbC without losing the benefits of a constructive program construction approach.

As future work, user studies could be conducted with all three approaches to further evaluate the usability of the approaches. We want to investigate how the more flexible construction approaches of TRAITCBC and CBC-BLOCK are received by developers. We also want to compare the development times and potential types of programming errors between the approaches. These user studies will help to develop concrete guidelines on which approach is appropriate under which circumstances and with which team.

**Acknowledgments.** This work was partially supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs (46.23.03). We thank Frederik Fröling for his work on CBC-BLOCK in his Master’s Thesis.

#### REFERENCES

- [ABB<sup>+</sup>16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. *Deductive Software Verification—The KeY Book: From Theory to Practice*, volume 10001. Springer, 2016.
- [ABD<sup>+</sup>14] Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Verification of Concurrent Systems with VerCors. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, volume 8483 of *Lecture Notes in Computer Science*, pages 172–216. Springer, 2014.
- [ABH<sup>+</sup>10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.
- [Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
- [Bac09] Ralph-Johan Back. Invariant Based Programming: Basic Approach and Teaching Experiences. *Formal Aspects of Computing*, 21(3):227–244, 2009.
- [Bar84] Hendrik P Barendregt. *The Lambda Calculus*, volume 3. North-Holland Amsterdam, 1984.
- [Bar03] John Gilbert Presslie Barnes. *High Integrity Software: The Spark Approach to Safety and Security*. Pearson Education, 2003.
- [BCD<sup>+</sup>05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *International Symposium on Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.
- [BCK<sup>+</sup>22] Tabea Bordis, Loek Cleophas, Alexander Kittelmann, Tobias Runge, Ina Schaefer, and Bruce W. Watson. Re-CorC-ing KeY: Correct-by-Construction Software Development Based on KeY. In *The Logic of Software. A Tasting Menu of Formal Methods*. Springer, 2022.
- [BDS10] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Implementing Software Product Lines Using Traits. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2096–2102, 2010.
- [BDSS13] Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocchio. TRAITRECORDJ: A Programming Language with Traits and Records. *Science of Computer Programming*, 78(5):521–541, 2013.
- [BEM07] Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. Testing and Verifying Invariant Based Programs in the SOCOS Environment. In *International Conference on Tests and Proofs (TAP)*, volume 4454 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2007.

- [BFL<sup>+</sup>11] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and Verification: The Spec# Experience. *Communication of the ACM*, 54(6):81–91, June 2011.
- [BLS04] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.
- [BMN14] Viviana Bono, Enrico Mensa, and Marco Naddeo. Trait-Oriented Programming in Java 8. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 181–186, 2014.
- [BRS20] Tabea Bordis, Tobias Runge, and Ina Schaefer. Correctness-by-Construction for Feature-Oriented Software Product Lines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 22–34. ACM, 2020.
- [BW12] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer Science & Business Media, 2012.
- [CDH<sup>+</sup>09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.
- [CDK<sup>+</sup>15] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. Synthesising Interprocedural Bit-Precise Termination Proofs. In *International Conference on Automated Software Engineering (ASE)*, pages 53–64. IEEE, 2015.
- [CDM13] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In *Certified Programs and Proofs: Third International Conference*, pages 147–162. Springer, 2013.
- [Cha06] Roderick Chapman. Correctness by Construction: A Manifesto for High Integrity Software. In *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*, SCS '05, pages 43–46, 2006.
- [CKK<sup>+</sup>12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *International Conference on Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.
- [Cok11] David R Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In *NASA Formal Methods Symposium*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer, 2011.
- [DDJS14] Ferruccio Damiani, Johan Dovland, Einar Broch Johnsen, and Ina Schaefer. Verifying Traits: An Incremental Proof System for Fine-Grained Reuse. *Formal Aspects of Computing*, 26(4):761–793, 2014.
- [Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communication of the ACM*, 18(8):453–457, August 1975.
- [Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [DNS<sup>+</sup>06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.
- [GJTV10] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Component Based Synthesis Applied to Bitvector Programs. Technical report, Citeseer, 2010.
- [GPS<sup>+</sup>17] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program Synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.
- [Gri87] David Gries. *The Science of Programming*. Springer, 1987.
- [Hal05] Robert J Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering*, 12(1):41–79, 2005.
- [HC02] Anthony Hall and Roderick Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software*, 19(1):18–25, 2002.
- [HKKN13] Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. Data Refinement in Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 100–115. Springer, 2013.
- [HL22] Maximilian PL Haslbeck and Peter Lammich. For a few dollars more: Verified fine-grained algorithm analysis down to llvm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(3):1–36, 2022.



- [Hoa71] Charles Antony Richard Hoare. Procedures and Parameters: An Axiomatic Approach. In *Symposium on Semantics of Algorithmic Languages*, pages 102–116. Springer, 1971.
- [IPW01] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.
- [JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. A Quick Tour of the VeriFast Program Verifier. In *Asian Symposium on Programming Languages And Systems*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer, 2010.
- [KFFD86] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.
- [KRMJ16] Mansur Khazeev, Victor Rivera, Manuel Mazzara, and Leonard Johard. Initial Steps Towards Assessing the Usability of a Verification Tool. In *International Conference in Software Engineering for Defence Applications*, volume 717 of *Advances in Intelligent Systems and Computing*, pages 31–40. Springer, 2016.
- [KRS20] Alexander Knüppel, Tobias Runge, and Ina Schaefer. Scaling Correctness-by-Construction. In *International Symposium on Leveraging Applications of Formal Methods*, pages 187–207. Springer, 2020.
- [KW12] Derrick G Kourie and Bruce W Watson. *The Correctness-by-Construction Approach to Programming*. Springer Science & Business Media, 2012.
- [LBR98] Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, pages 404–420. Citeseer, 1998.
- [Lei95] K Rustan M Leino. *Toward Reliable Modular Programs*. California Institute of Technology, 1995.
- [Lei10] K Rustan M Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.
- [LS08] Luigi Liquori and Arnaud Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):1–32, 2008.
- [LT12] Peter Lammich and Thomas Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft’s Algorithm. In *Interactive Theorem Proving: Third International Conference, ITP 2012*, pages 166–182. Springer, 2012.
- [LW94] Barbara H Liskov and Jeannette M Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.
- [Mey88] Bertrand Meyer. Eiffel: A Language and Environment for Software Engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.
- [Mey92] Bertrand Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, 1992.
- [Mor94] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 2nd edition, 1994.
- [MW80] Zohar Manna and Richard Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, jan 1980.
- [OCW03] Marcel Vinicius Medeiros Oliveira, Ana Cavalcanti, and Jim Woodcock. ArcAngel: A Tactic Language for Refinement. *Formal Aspects of Computing*, 15(1):28–47, 2003.
- [OCW09] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP Semantics for Circus. *Formal Aspects of Computing*, 21(1):3–32, 2009.
- [OGC08] Marcel Vinicius Medeiros Oliveira, Alessandro Cavalcante Gurgel, and C G Castro. CRefine: Support for the Circus Refinement Calculus. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 281–290. IEEE, Nov 2008.
- [PG13] David J Pearce and Lindsay Groves. Whiley: A Platform for Research in Software Verification. In *International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 238–248. Springer, 2013.
- [PKSL16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. *ACM SIGPLAN Notices*, 51(6):522–538, aug 2016.
- [PR01] Malte Plath and Mark Ryan. Feature Integration Using a Feature Construct. *Science of Computer Programming*, 41(1):53–84, 2001.

- [RBTS21] Tobias Runge, Tabea Bordis, Thomas Thüm, and Ina Schaefer. Teaching Correctness-by-Construction and Post-hoc Verification—The Online Experience. In *Formal Methods Teaching Workshop*, volume 13122 of *Lecture Notes in Computer Science*, pages 101–116. Springer, 2021.
- [RKS<sup>+</sup>22] Tobias Runge, Alexander Kittelmann, Marco Servetto, Alex Potanin, and Ina Schaefer. Information Flow Control-by-Construction for an Object-Oriented Language. In *International Conference on Software Engineering and Formal Methods*, volume 13550 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2022.
- [RKTS20] Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. Lattice-Based Information Flow Control-by-Construction for Security-by-Design. In *FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering*, pages 44–54. ACM, 2020.
- [RPTS22] Tobias Runge, Alex Potanin, Thomas Thüm, and Ina Schaefer. Traits: Correctness-by-Construction for Free. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, volume 13273 of *Lecture Notes in Computer Science*, pages 131–150. Springer, 2022.
- [RSC<sup>+</sup>19] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick Kourie, and Bruce W Watson. Tool Support for Correctness-by-Construction. In *International Conference on Fundamental Approaches to Software Engineering*, volume 11424 of *Lecture Notes in Computer Science*, pages 25–42. Springer, 2019.
- [RTC<sup>+</sup>19] Tobias Runge, Thomas Thüm, Loek Cleophas, Ina Schaefer, and Bruce W Watson. Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study. In *Formal Methods. FM 2019 International Workshops. Refine*, volume 12233 of *Lecture Notes in Computer Science*, pages 388–405. Springer, 2019.
- [SD05] Charles Smith and Sophia Drossopoulou. Chai: Traits for Java-Like Languages. In *European Conference on Object-Oriented Programming*, pages 453–478, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [SFS12] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-Based Function Summaries in Bounded Model Checking. In *Hardware and Software: Verification and Testing*, volume 7261 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2012.
- [SH19] Dominic Steinhöfel and Reiner Hähnle. Abstract Execution. In *International Symposium on Formal Methods*, pages 319–336. Springer, 2019.
- [STAL11] Wolfgang Scholz, Thomas Thüm, Sven Apel, and Christian Lengauer. Automatic Detection of Feature Interactions Using the Java Modeling Language: An Experience Report. In *Proceedings of the 15th International Software Product Line Conference, Volume 2, SPLC '11*, New York, NY, USA, 2011. Association for Computing Machinery.
- [SWL<sup>+</sup>94] Mark Stickel, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood. Deductive Composition of Astronomical Software from Subroutine Libraries. In *International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 1994.
- [tBCSW18] Maurice H. ter Beek, Loek Cleophas, Ina Schaefer, and Bruce W. Watson. X-by-Construction. In *International Symposium on Leveraging Applications of Formal Methods*, pages 359–364, Cham, 2018. Springer International Publishing.
- [TFNP15] Julian Tschannen, Carlo A Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 566–580. Springer, 2015.
- [TSAH12] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-Based Deductive Verification of Software Product Lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering, GPCE '12*, page 11–20, New York, NY, USA, 2012. Association for Computing Machinery.
- [WKSC16] Bruce W. Watson, Derrick G. Kourie, Ina Schaefer, and Loek Cleophas. Correctness-by-Construction and Post-hoc Verification: A Marriage of Convenience? In *International Symposium on Leveraging Applications of Formal Methods*, volume 9952 of *Lecture Notes in Computer Science*, pages 730–748. Springer, 2016.
- [WRH<sup>+</sup>12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.

- [ZOC09] Frank Zeyda, Marcel Oliveira, and Ana Cavalcanti. Supporting ArcAngel in ProofPower. *Electronic Notes in Theoretical Computer Science*, 259:225–243, 2009.

#### APPENDIX A. INTERVIEW QUESTIONS

- (1) Which task was more difficult and why?
- (2) Which tasks were solved?
- (3) What were the biggest problems during the development?
- (4) Is the development according to CbC understandable?
- (5) Is the use of the block rules understandable?
- (6) Is the introduction of the block rules reasonable?
- (7) Would you use the block rules when implementing according to CbC?
- (8) How do you like the development in the textual editor?
- (9) How do you like the development in the graphical editor?
- (10) Is the textual or the graphical editor preferred?
- (11) Which elements from the editors are particularly helpful or inadequate and why?
- (12) What functionalities are still missing in the editors?
- (13) What would it take for you to develop according to CbC in your workday?

## **A.6. Immutability and Encapsulation for Sound OO Information Flow Control**

# Immutability and Encapsulation for Sound OO Information Flow Control

TOBIAS RUNGE, Karlsruhe Institute of Technology, Institute of Information Security and Dependability (KASTEL), Germany and TU Braunschweig, Institute of Software Engineering and Automotive Informatics, Germany

MARCO SERVETTO, Victoria University of Wellington, New Zealand

ALEX POTANIN, Australian National University, Australia

INA SCHAEFER, Karlsruhe Institute of Technology, Institute of Information Security and Dependability (KASTEL), Germany and TU Braunschweig, Institute of Software Engineering and Automotive Informatics, Germany

Security-critical software applications contain confidential information which has to be protected from leaking to unauthorized systems. With language-based techniques, the confidentiality of applications can be enforced. Such techniques are for example type systems that enforce an information flow policy through typing rules. The precision of such type systems, especially in object-oriented languages, is an area of active research: an appropriate system should not reject too many secure programs while soundly preserving noninterference. In this work, we introduce the language SIFO which supports information flow control for an object-oriented language with type modifiers. Type modifiers increase the precision of the type system by utilizing immutability and uniqueness properties of objects for the detection of information leaks. We present SIFO informally by using examples to demonstrate the applicability of the language, formalize the type system, prove noninterference, implement SIFO as a pluggable type system in the programming language L42, and evaluate it with a feasibility study and a benchmark.

CCS Concepts: • **Security and privacy** → **Information flow control**.

Additional Key Words and Phrases: security, information flow, type system, mutation control, confidentiality, integrity

## 1 INTRODUCTION

In security-critical software development, it is important to guarantee the confidentiality and integrity of the data. For example, in a client-server application, the client has a lower privilege than the server. If the client reads information from the server in an uncontrolled manner, we may have a violation of confidentiality; this causes the client to release too much information to the user. On the other hand, if the server reads information from the client in an uncontrolled manner, we may have a violation of integrity; this causes the server to accept input that has not been validated.

Language-based techniques such as type systems are used to ensure specific information flow policies for confidentiality or integrity [Sabelfeld and Myers 2003]. A type system assigns an explicit security type to every variable and expression, and typing rules prescribe the allowed information flow in the program and reject programs violating the security policy. For example, we can define a

---

Authors' addresses: Tobias Runge, tobias.runge@kit.edu, Karlsruhe Institute of Technology, Institute of Information Security and Dependability (KASTEL), Germany, TU Braunschweig, Institute of Software Engineering and Automotive Informatics, Germany; Marco Servetto, marco.servetto@ecs.vuw.ac.nz, Victoria University of Wellington, New Zealand; Alex Potanin, alex.potanin@anu.edu.au, Australian National University, Australia; Ina Schaefer, ina.schaefer@kit.edu, Karlsruhe Institute of Technology, Institute of Information Security and Dependability (KASTEL), Germany, TU Braunschweig, Institute of Software Engineering and Automotive Informatics, Germany.

---

© 2022 Association for Computing Machinery.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Programming Languages and Systems*, <https://doi.org/10.1145/3573270>.

security policy as a lattice of security levels with the highest level **high** and the lowest level **low**, and an information flow from **high** to **low** is prohibited.

For simple while-languages, type systems to control the information flow are widely studied [Hunt and Sands 2006; Li and Zhang 2017; Volpano et al. 1996]. We focus on the less researched area of information flow control for object-oriented languages. Analysis techniques such as fine-grained taint analysis [Arzt et al. 2014; Enck et al. 2014; Graf et al. 2013; Hedin et al. 2014; Huang et al. 2014, 2012; Milanova and Huang 2013] detect insecure flows from sources to secure sinks by analyzing the flow of data in the program. Coarse-grained dynamic information flow approaches [Jia et al. 2013; Nadkarni et al. 2016; Roy et al. 2009; Xiang and Chong 2021] reduce the writing effort of annotations by tracking information at the granularity of lexically or dynamically scoped section of code instead of program variables. By writing annotation, users can increase precision of the information flow results [Xiang and Chong 2021]. Moreover, there are approaches using program logic [Amtoft et al. 2006, 2008; Beckert et al. 2013] to analyze and reason about information flow.

In this work, we focus on security type systems for object-oriented languages [Banerjee and Naumann 2002; Barthe et al. 2007; Myers 1999; Strecker 2003]. Sun, Banerjee, and Naumann [Banerjee and Naumann 2002; Sun et al. 2004] created a Java-like language annotated with security levels for the standard information flow policy with only two security levels. Myers et al. [Myers 1999] created the Jif language which extends Java with a type system that supports information flow control. The precision of the type systems for object-oriented languages is a major challenge. Both related approaches do not have an alias analysis or an immutability concept, so they conservatively reject secure programs where confidential and non-confidential references could alias the same object. This important drawback is addressed in our work. Additionally, as done for other type systems, we give a correctness guarantee through a proof of noninterference: **high** data never influences **low** data. This means that an attacker who can observe **low** data cannot obtain information about **high** data. If an untrusted library is in the code base, the developer can leverage the type system to ensure that only **low** data is served to such library.

We introduce SIFO<sup>1</sup> which supports information flow control for an object-oriented language with type modifiers for mutability and alias control [Giannini et al. 2019]. With respect to former work on security type systems for object-oriented languages, SIFO provides a more precise type system, allowing to type more correct programs. In this work, we show that reasoning about immutability and encapsulation is beneficial to reason about information flow. In addition to adding expressivity, SIFO allows a natural and compact programming style, where only a small part of the code needs to actually be annotated with security levels. This result is achieved by building over the concept of promotion/recovery [Giannini et al. 2019; Gordon et al. 2012], and extending it to allow methods and data structures to be implicitly parametric on the security level. For example, with promotion, a data structure can be used with any security level, but security is still enforced by not allowing data structures of different security levels to interfere with each other. This reduces the programming effort of developers and supports reuse of programs and libraries [Giannini et al. 2019].

The contents of this paper are as follows. First, we introduce the language SIFO for information flow control. Second, we formalize the type system by introducing typing and reduction rules. Third, we show that our language is sound by proving the noninterference property that secret data is never observable by a public state. Fourth, we implement SIFO and evaluate it with a feasibility study and a benchmark to compare SIFO with state-of-the-art information flow analysis tools.

---

<sup>1</sup>SIFO is an acronym for Secure Information Flow in an Object-oriented language

## 2 INFORMAL PRESENTATION OF SIFO

In this section, we explain the challenges of securely checking the information flow in object-oriented languages. We then give an informal introduction to SIFO. Last, we discuss well and ill typed SIFO expressions for a more detailed explanation.

### 2.1 Motivating example

Consider the following partially annotated code using two security levels **low** and **high**:

```

1 class Person { low id; Person(low id){ this.id=id; } }
2 ...
3 low local_id = GiveMe.anId();
4 high p = new Person(local_id);
5 high inside = p.id;

```

In SIFO, security is an instance based property: the person  $p$  is **high**, but other persons could have a different security level. Security is also a deep property: the content of all the fields of  $p$  encodes **high** information. In our example, every person has an `id`. Even if the `id` field is declared **low**, it will encode **high** information inside of the **high** instance  $p$ . The value of `local_id` is **low**. We can use it to initialize `id` since information can flow from **low** to **high**, but not from **high** to **low**.

When extracting the value of the field `id`, the information is now part of the **high** Person, and thus, needs to be seen as **high**:  $p.id$  produces a **high** value.

Is this code conceptually correct with respect to information flow? Can we complete the type annotations on this code to make it correct? If the `id` is just a primitive integer, this is possible and easy in both SIFO<sup>2</sup> and other languages for information flow, such as Jif [Myers 1999]:

```

1 class Person { //SIFO CODE
2   Int id; //low is the default security level
3   Person(Int id){ this.id = id; } }
4 ...
5 local_id = GiveMe.anId(); //a low Int
6 high p = new high Person(local_id);
7 high inside = p.id;

```

The corresponding Jif code is also quite easy, but a little more involved and with a different syntax; we report it in Listing 1. Both in SIFO in Line 6, and in Jif in Line 9, the value of `local_id` gets *promoted* from **low** to **high**.

```

1 class Person[label L] {
2   final int{L;this} id;
3   Person(int{L;this} id){ this.id = id; }
4 }
5 ...
6 //label {high->low} allows low and high to read the variable
7 int{high->low} id = GiveMe.anId();
8 //label {high->} allows only high to read the variable
9 Person[{high->}]{high->} p = new Person[{high->}](id);
10 int{high->} inside = p.id;

```

Listing 1. Example in Jif syntax

What happens if the `id` is a more complex custom object type? The following code is accepted in SIFO:

<sup>2</sup>In the examples, we use a rich language including local variables and literals with the usual semantics. Those are supported by our artifact, but in the formal model we present a minimal language where we keep only the most crucial OO features.



```

1 class Account { Int id; String name; Date firstTransaction;... }
2 class Person {
3   Account id;
4   Person(Account id){ this.id = id; } }
5 ...
6 local_id = GiveMe.anId();//a low Account
7 high p = new high Person(local_id);
8 high inside = p.id;
9 high name = inside.name;

```

As you can see, not much has changed. Of course, we need to define the `Account` class, but then we can use it in the same way we use `Int` before.

On the other hand, Jif [Myers 1999] (the most closely related work) cannot accept this kind of code. In a pure object-oriented setting, everything is an object, and pre-defined types, as integers, should be treated as any other object. However, Jif treats primitive types in a privileged way. In Jif, it is possible to write more flexible code relying on primitive types than on objects. The difficulty revolves around aliasing and mutation: the local variable `local_id` is still available, and now it is aliased inside of the `high` `Person` object `p`. Thus, if `p` is used to update any field of the `Account`, then the `low` part of the program could see this `high` information through `local_id`. This can happen because with the base type system of Java all objects can be both mutated and aliased. Jif builds on top of Java and only adds type properties directly related to information flow, so it cannot make immutability and aliasing assumptions. On the other side, SIFO builds on top of L42 [Giannini et al. 2019], a language with built-in support for immutability and aliasing control using type modifiers (also called reference capabilities).

In L42, the default modifier for references is `imm` (immutable), and the default security level of SIFO is `low`. Thus, a fully annotated version of the code above would look as follows:<sup>3</sup>

```

1 class Account {
2   low imm Int id; low imm String name; low imm Date firstTransaction;
3   ... }
4 class Person {
5   low imm Account id;
6   Person(low imm Account id){ this.id = id; }
7   }
8   ...
9 low imm Account local_id = GiveMe.anId();
10 high mut Person p = new high Person(local_id);
11 high imm Account inside = p.id;
12 high imm String name=inside.name;

```

Since the `Account` is immutable, in SIFO the value of `local_id` is promoted from `low` to `high` for the constructor call, exactly as it happens for `Int` before. Indeed, deeply immutable objects allow for the same kind of reasoning that primitive types allow in Java. In this way, SIFO code can scale and use objects as easily as primitive types in contrast to Jif and other approaches.

To make the same kind of behavior accepted in Jif, the code would have to be modified in the following way:

```

1 low mut Account local_id = GiveMe.anId(); //in Jif, there is no immutability
2 low mut Date a = local_id.firstTransaction;

```

<sup>3</sup>To help readability, in the rest of the paper we will write type modifiers and security levels explicitly, but a syntactically much lighter style, as shown above, is accepted by our artifact and it is the preferred way to code, once the programmer gets used to those defaults.

```

3  high mut Person p = new high Person(
4    new high Account(local_id.id, local_id.name,
5    new high Date(a.day, a.month, a.year)));

```

This code is accepted by both SIFO and Jif. This is a technique called *defensive cloning* [Bloch 2016]; it is very popular in settings where aliasing and mutability cannot be controlled.

In SIFO, we have *mutable* and *immutable* objects; where the reachable object graph (ROG) of an immutable object is composed only of other immutable objects (deep immutability), while the ROG of a mutable object can contain both mutable and immutable objects [Giannini et al. 2019]. The set of mutable objects in a ROG is called MROG.

In addition to *imm*, L42 also offers the *capsule* concept: a capsule reference refers to a mutable object whose MROG is reachable only through such reference. Both *imm* and *capsule* references can be safely promoted from *low* to *high*; this avoids the need of defensive cloning also when encapsulated mutable state is involved.

```

1  low capsule Account local_id = GiveMe.anId();
2  high mut Person p = new high Person(local_id);

```

Capsule variables are affine, that is, they can only be used zero or one time, thus if *p* is used to update the state of the account, the local capsule variable *local\_id* cannot be used to examine these updates.

As you can see from those examples, aliasing and mutability control is a fundamental tool needed to support information flow in the context of an object-oriented language. A typical misunderstanding of type modifiers is that a mutable field would always refer to a mutable object. This is not the case, indeed all the fields of immutable objects will transitively contain only immutable objects. This of course includes all fields originally declared as mutable. The same applies to security labels: a *low* field in a *high* object would transitively contain only *high* objects. This is different with respect to many other object-oriented languages, where the declarations determine what to expect. If there is information from the context, that is normally explicit in the usage site. In SIFO instead, the declared type is only a first approximation: the security level (and type modifier) of an expression is a combination of what is declared in the class table and what is implied from the usage site.

Note how in our example the class *Person* is declared with a *low* field, but the *high* *Person* object actually stores a *high* value for such field. In SIFO, a *low* field is not like a *low* field in Jif, but it is more like a field with generic/parametric security: the value of a *low* field of a *low* object will be *low* but the value of a *low* field of a *high* object will be *high*. In general, the ROG of an object is always at least as secure as the security of the object itself. This aligns nicely with mutability control in L42, where immutable instances of classes declaring a *mut* field will hold immutable values in such fields. Deep properties (like L42 immutability and SIFO security) allow for a much simpler and more predictable reasoning with respect to (optionally) shallow properties, like Rust immutability (that supports internal mutability) and Jif security (where *low* values can be stored in the ROG of *high* values).

## 2.2 SIFO Concepts

*Objects and references.* As we anticipated above, in SIFO, we have *mutable* and (deeply) *immutable* objects. We also have four kinds of references: *imm*, *mut*, *capsule*, and *read* [Giannini et al. 2019]. An *imm* reference must point to an immutable object, and can be freely aliased, but as the name suggests, the fields of an immutable object cannot be updated. A *mut* reference must point to a mutable object; such an object can be aliased and mutated. A *capsule* reference points to a mutable object that is kept under strict aliasing control: the mutable ROG reachable from the *capsule* reference cannot be

$$\begin{aligned}
T & ::= s \text{ mdf } C \\
s & ::= \text{high} \mid \text{low} \mid \dots \text{ (user defined)} \\
\text{mdf} & ::= \text{mut} \mid \text{imm} \mid \text{capsule} \mid \text{read} \\
CD & ::= \text{class } C \text{ implements } \overline{C} \{ \overline{F} \overline{MD} \} \mid \text{interface } C \text{ extends } \overline{C} \{ \overline{MH} \} \\
F & ::= s \text{ mut } C f; \mid s \text{ imm } C f; \\
MD & ::= MH \{ \text{return } e; \} \\
MH & ::= s \text{ mdf method } T m(T_1 x_1, \dots, T_n x_n) \\
e & ::= x \mid e.f \mid e_0.f = e_1 \mid e_0.m(\overline{e}) \mid \text{new } s C(\overline{e})
\end{aligned}$$

Fig. 1. Syntax of the core calculus of SIFO

reached from other references. The **capsule** reference can be used only once to assign this isolated portion of the heap to a reference of any kind. In particular, this means that a **capsule** reference can be used to initialize/update an **imm** reference/field; when this happens all the objects in the ROG of such a reference become immutable.

The “only used once” restriction is necessary so that no alias for the isolated portion of the heap can be introduced, which would violate the **capsule** property.

Finally, a **read** reference is the common supertype of **imm** and **mut**. With a **read** reference, the ROG cannot be mutated and aliases cannot be created; but there is no immutability guarantee that the object is not accessible by other references, even **mut** ones.

*Types.* Types in SIFO are composed by a security level  $s$ , a type modifier  $\text{mdf}$ , and a class name  $C$ . The security levels  $s$  are arranged in a lattice that specifies the allowed data flow direction between them. For example, we have a lattice with a **low** and a **high** security level, where the allowed information flow is from **low** to **high**, but not vice versa. The type modifier  $\text{mdf}$  can be **mut**, **imm**, **capsule**, and **read** as introduced above. The subtyping relation between modifiers is defined as follows: for all  $\text{mdf}$ ,  $\text{capsule} \leq \text{mdf}$ ,  $\text{mdf} \leq \text{read}$ . This means that, for example, a **mut** reference can be used if a **read** is needed, and a **capsule** reference can be used both as a **mut** or as an **imm** one. This is sound, because **capsule** variables can be used only once.

*Core Calculus.* The syntax of the core calculus of SIFO is shown in Fig. 1. It covers classes  $C$ , field names  $f$ , method names  $m$ , and declarations for classes, interfaces, and methods. A class consists of fields and methods. The class itself has no modifier or security level. The modifiers and security levels are associated with references and expressions. A field has a type  $T$  and a name. A method has a return type, a list of input parameters with names and types, and also a security level and a type modifier for the receiver; they are specified in front of the keyword **method**. We have the standard expressions: variable, field access, field assignment, method call, and object construction. When an object is initialized, its security level is initially determined by the constructor invocation. Thus, different references to objects of the same class can have different security levels.  $C$ ,  $m$ ,  $f$ , and  $x$  in Fig. 1 are all disjoint syntactic categories.

*Method Calls.* A method has to be defined in a class with parameter types, a return type and a receiver type. For example, an `Accumulator` class has a **low** `Int` field `acc` and a method `add` that adds another **low** `Int` parameter to the `acc` field and updates the field.

```

1 class Accumulator {
2   low imm Int acc;
3   low mut method low imm Int add(low imm Int x){ return this.acc=this.acc+x; }
4 }
5 ...// below we show examples of user code

```

```

6 low Accumulator a = GiveMe.aLowAcc();//a low Accumulator
7 low imm Int w = 5;
8 low imm Int x = a.add(w); //ok to call as low*low -> low
9
10 high Accumulator b = GiveMe.aHighAcc();//a high Accumulator
11 high imm Int y = 6;
12 high imm Int z = b.add(y); //ok to call as high*high -> high

```

We can call such a method if the receiver and the actual input parameter are **low**. However, SIFO adds flexibility to method calls using *multiple method types*. Formally defined in Section 4, they allow to call a method as if it was declared with a range of different type modifiers and security levels. In this example, the multiple method types rule allows us to call the `add` method also if the receiver and the parameter are all increased to the same security level (for example to **high**) and returning a value with this same security level. Without this feature, the method `add` needs to be declared for each security level. This feature also has benefits in comparison to a parameterized version of the language because legacy code and standard libraries can be used in SIFO without adding security level annotations: **low** is the default security level, and **imm** is the default modifier.

*Control Flow and Implicit Information Leaks.* Information flow control mechanisms [Sabelfeld and Myers 2003; Volpano et al. 1996] are used to enforce an information flow policy that specifies the allowed data flow in programs. A program can leak information *directly* through a field update. This can be prevented by ensuring that no confidential data is assigned to a less confidential variable. However, information can also flow *implicitly* through conditionals, loops, and (crucial in OO) dynamic dispatch. For example, the chosen branch of a conditional reveals information about the values in the guard. As shown from Smalltalk [Goldberg and Robson 1983], in a pure OO language, dynamic dispatch can be used to emulate conditional statements and various forms of iterations and control flow. Thus, our core language does not contain explicit conditional statements, but they can be added as discussed in Section 4. Loops can be implemented through recursive method calls.

Therefore, SIFO only needs a secure method call rule to prevent implicit information flow leaks. In a method call, information of the method receiver can flow to the return value and mutable parameters. Thus, the security levels of the return value and mutable parameters have to be equal or higher than the security level of the receiver. Consider for example the following code: `res=myValue.aOrB(a,b)` If the method `aOrB` returns the first or the second parameter depending on the dynamic type of `myValue`, we could use the result to identify information about `myValue`. Note how this pattern is very similar to the Church encoding of Booleans. Similarly, if parameter `a` is typed **low mut** and the receiver has a **high** security level, information of the **high** receiver can be leaked by observing the mutations on the parameter `a` after the method call.

### 2.3 Examples of Well-Typed and Ill-Typed SIFO Expressions

In Listing 2, we show secure and insecure programming statements to explain the reasoning about information flow in SIFO.

A class `Person` contains a **low imm** `String` `name` and two **high** fields: a **mut** `Password` and an **imm** `AccountId`. The `Password` and `AccountId` class have a `String` field to set the actual password/id. When accessing a field, we consider the security level of both the field and the receiver and determine the least upper bound of both security levels in the lattice. When an object is initialized, it is created as mutable, and the initial security level is determined by the constructor invocation. For example, a `Person` can be initialized with a **low** or with a **high** security level.

Consider the assignments in Listing 2 starting with Line 5 (line numbers are referenced in parentheses in the following): To ensure confidentiality, the type system prevents the password to

```

1  class Person{low imm String name;  high mut Password pwd;  high imm AccountId id;}
2  class Password{low imm String pwdS;}
3  class AccountId{low imm String idS;}
4
5  low mut Person p =...//a pre existing low Person reference
6  high mut Password pass = p.pwd;//ok, access of high Password
7  high imm String passS = p.pwd.pwdS;//ok, access is typed high
8  low imm String passS = p.pwd.pwdS;//wrong, high assigned to low
9
10 p.pwd.pwdS = highString;//ok, field update with high String
11 p.pwd.pwdS = p.name;//ok, as an imm String can be promoted
12 p.name = highString;//wrong, high String assigned to low p.name
13 high mut Person pHigh =...//a pre existing high Person
14 pHigh.name = highString;//ok, field update with high String
15
16 low mut Password newPass = new low Password("some");//ok
17 p.pwd = newPass;//wrong, mutable secret shared as low and high
18 newPass.pwdS = "password";//ok? Insecure with previous line
19
20 low capsule Password capsPass=new low Password("secret");//ok
21 p.pwd = capsPass;//ok, no alias introduced
22
23 low imm AccountId aid = new low AccountId("secretId");//ok
24 p.id = aid;//ok, aid is imm and can be aliased
25 aid.idS = "0";//wrong, immutable object cannot be updated
26
27 low read Person pRead = p;//ok, assigned to read reference
28 high imm String passS = pRead.pwd.pwdS;//ok, access is high
29 pRead.pwd.pwdS = highString;//wrong, read cannot be updated
30 someMutObject.fieldName = pRead;//wrong; there are no read fields
31 someMutObject.fieldName = pRead.id;//ok if the field has an imm type

```

Listing 2. SIFO examples

be leaked via a **low** reference (8), but it can be exposed to another **high** reference (6, 7). The password can be updated with another **high** String (10), or with a **low** String (11), as we allow to promote the security level of **imm** references. The opposite of updating a **low** field of a **low** reference with a **high** String is forbidden (12), but the assignment is allowed if the reference and the String are **high** (14).

Until now, we explained assignments of immutable Strings; but the most interesting challenge to guarantee confidentiality is about assigning mutable objects instead. For example, how can we update the mutable `p.pwd` field? When a new `Password` object is created, it can be initialized as a **low** object as the `Password` is not confidential on its own. The confidentiality is the association between the `Person` object and the `Password` object. SIFO prevents that a **low** reference to a `Password` object is assigned to a `Person` object (17). The reason is that the variable `newPass` is still in scope after the field update, thus if (17) was accepted, (18) could be used to sneak a password change without the need of any **high** information.

A secure assignment without aliases is shown in (20, 21). Here, the **capsule** modifier is utilized. A reference to a `Password` object can be assigned to a `Person` object, if the reference to the `Password` object is **high**. The password is initialized by the programmer as **low capsule**. The system of type modifiers is flexible enough to promote the created object from **mut** to **capsule**. Since there is no **mut**



value in the input, we are sure that an isolated portion of memory is created, as the created object cannot be accessed from any other `mut` reference. In (21), the flexible type system can then promote the variable `capsPass` to a **high capsule**, which is assigned to the `p.pwd` field.

As discussed before, aliases over `imm` references (24), are allowed to move from lower security to higher one. The alias does not lead to a security leak because the type system ensures that fields of `AccountId aid` cannot be updated (25). Both `imm` and `capsule` references are referentially transparent, and can be used as a controlled way to communicate between different security levels.

Finally, with `read` references (27), `imm` fields can still be accessed (28), but no fields can be updated (29). Here, we present a simplified L42 type system, where fields can only be `mut` or `imm`; thus there is no field that can be updated using a `read` reference (30). In the full L42, it is possible to have, for example, `read` linked lists of `read` elements, but this has some subtle interaction with promotions, so we omit it here for simplicity. Of course, `imm` references reached from `read` references (31) can be assigned to `imm` fields as usual.

For a more compelling example of our system that can promote expressions, consider the following listing:

```

1  class PassFactory{
2  ...
3  low imm method low mut Password from(low imm String base){
4      low mut Password res = new low Password(base);
5      if (this.tooSimple(base)){res.pwd = this.complete(base);}
6      return res;
7  }
8  }
9  ...
10 p.pwd = passFactory.from("foo")

```

The method `from` is well typed. The method `from` returns a `low mut Password res` which could not be directly assigned in (10) because a **high** security level is needed, but the system of type modifiers is flexible enough to promote `low mut Password res` to a **high** security level by utilizing the `capsule` modifier. Since there is no `mut` value in the input, we are sure that an isolated portion of memory is created (a `capsule`). With controlled aliasing, we can promote the `capsule` reference to a **high** security level (i.e., a `low capsule Password` can be promoted to a `high capsule Password`), and then, it can be assigned to the field in (10). All in all, a `mut` references can be promoted to a `capsule`, transferred to another security level and then reassigned to another `mut` reference.

Any method that takes a single `mut` in input, mutates it, and returns it as `mut` can be called with a `capsule` parameter and the result will also be promoted to `capsule`. This pattern allows great flexibility when encapsulated mutable objects need to be mutated [Giannini et al. 2019].

### 3 DEFINITIONS FOR THE SIFO TYPE SYSTEM

In this section, we define well-formedness of the type system and useful helper methods to introduce typing rules in the following section.

*Well-Formedness.* A well-formed program respects the following conditions: All classes and interfaces are uniquely named. All methods in a specific class or interface are uniquely named. All fields in a specific class are uniquely named. All parameters in a method header are uniquely named, and there is no explicit method parameter called `this`. The subtyping graph induced by implemented interfaces is acyclic (in this simplified language, we do not have class extension). `capsule` references can be used at most one time.

- $sec(T) = s$ , returns the security level  $s$  in type  $T$ .
- $mdf(T) = mdf$ , returns the modifier in type  $T$ .
- $class(T) = C$ , returns the class  $C$  in the type  $T$ .
- $fields(C) = T_1 f_1 \dots T_n f_n$ , returns the field declarations of class  $C$ .
- $p(C) = \text{class } C \text{ implements } \overline{C} \{ \overline{F} \overline{M} \}$ , returns the declaration of class  $C$ .
- $lub(s_0, \dots, s_n) = s$ , returns the least upper bound of the parameters  $s_0, \dots, s_n$ .
- $T[s'] = lub(s, s') mdf C$ , where  $T = s mdf C$ , defined only if  $s' \leq s$  or  $s \leq s'$ ; returns a new type with security level  $lub(s, s')$
- $mdf \triangleright mdf' = mdf''$ , returns the modifier of an expression when accessing a field.
  - $mut \triangleright mdf = capsule \triangleright mdf = mdf$
  - $imm \triangleright mdf = mdf \triangleright imm = imm$
  - $read \triangleright mut = read$ .

Fig. 2. Helper functions

If  $s mdf \text{ method } T m(T_1 x_1 \dots T_n x_n)$  is declared in  $C$ , with  $T_0 = s mdf C$  then

- 1:  $T_0[s'] \dots T_n[s'] \rightarrow T[s'] \in methTypes(C, m)$
- 2:  $(T_0[s'] \dots T_n[s'] \rightarrow T[s'])[mut \setminus capsule] \in methTypes(C, m)$
- 3:  $(T_0[s'] \dots T_n[s'] \rightarrow T[s'])[read \setminus imm, mut \setminus capsule] \in methTypes(C, m)$

Fig. 3. Definition of multiple method types

*Helper Functions.* In Figure 2, we show some helper functions for our type system. The first three notations extract the security level, the type modifier, and the class name from a type. The next two return fields and class declarations. The  $lub$  operator is defined to return the least upper bound of a set of input security levels arranged in a lattice. For example, since  $low \leq high$ , we have  $lub(low, high) = high$ . A lattice of security levels was first introduced by Bell and LaPadula [Bell and La Padula 1976], and Denning [Denning 1976]. A lattice is a structure  $\langle L, \leq, lub, \top, \perp \rangle$  where  $L$  is a set of security levels and  $\leq$  is a partial order (e.g.,  $low \leq high$ ). The lattice defines an upper bound of security levels. A set of elements  $X \subseteq L$  has an upper bound  $y$  if  $\forall x \in X : x \leq y$ . An upper bound  $u$  of  $X$  is the least upper bound ( $lub$ ) if  $u \leq y$  for each upper bound  $y$  of  $X$ . To form an upper semi-lattice, a unique least upper bound ( $lub$ ) for every subset of  $L$  must exist. Additionally, we restrict the lattice to be bounded with the greatest element  $\top$  and the least element  $\perp$ .

In Figure 2, the function  $s mdf C[s']$  returns a new type whose security level is the least upper bound of the two. The security level is set to  $lub(s, s')$  and the modifier and class remain the same. The last function  $mdf \triangleright mdf'$  computes a resulting modifier if a field with type modifier  $mdf'$  is accessed from some reference with type modifier  $mdf$ . For example, if we access a  $mut$  field from a  $mut$  reference we get a  $mut$  value, but if we access a  $mut$  field from a  $read$  reference, we get a  $read$  value. If either the reference or the field are  $imm$ , then  $imm$  is returned; thanks to deep immutability, the whole reachable object graph is immutable.

*Multiple Method Types.* Instead of a single method type as in Featherweight Java [Igarashi et al. 2001], we return a set of method types using  $methTypes(C, m) = \{ \overline{T_0} \rightarrow T_0, \dots, \overline{T_n} \rightarrow T_n \}$ . The programmer just declares a method with a single type, and the others are deduced by applying all the transformations shown in Fig. 3. Multiple method types reduce the need of implementing the same functionality several times, where the same parameter has only different type modifiers or security levels. The base case, as declared by the programmer, can be transformed in various ways: (1) A method working on lower security data can be transparently lifted to work on higher



security data (some security level  $s'$ ). This means that methods that are not concerned with security are usually declared as working on a **low** receiver and **low** parameters, returning a **low** result. Note that the security level remains unchanged when  $s'$  is chosen as **low**. Thus, we can use  $T[s']$  with  $s'$  different from **low** when we are in a context where we need to manipulate secure data. The multiple method types lift the method as if it was declared with higher receiver, parameters, and return type. For example, a mathematical method should return the same security level as the security level of the parameters. In our language, we can just implement this method once with the lowest security level and reuse it with any other security level of the lattice. As a comparison, the Jif tutorial<sup>4</sup> suggests that a mathematical method should be implemented with a generic security level.

(2) The second case swaps all **mut** types for **capsule** ones. If we provide **capsule** instead of **mut** in the input, we can use the method to produce a **capsule** return value. This corresponds to **capsule** recovery/promotion in [Giannini et al. 2019]. By providing all **mut** parameters as **capsule**, the method would not take any **mut** as input. Any **mut** object that is returned, is created inside of the method execution (as we do not have any form of global state/variables) and thus can be seen as a **capsule** from outside the scope of the method body. For example, a method declared as

```
low mut method low mut Person father(low imm String name)
```

can be also used as if it was declared as

```
low capsule method low capsule Person father(low imm String name),
```

where all **mut** parameters (just the receiver in this example) and the return type are turned into **capsule**. (3) The third case swaps all **mut** types for **capsule** ones and all **read** types for **imm** ones. This is useful if the method was returning a **read** value; in this case we can obtain an **imm**. This corresponds to immutable recovery/promotion in [Giannini et al. 2019] and can be intuitively understood by considering that a **read** reference can point to either an immutable or a mutable object. If it was an immutable object, it is fine to return it as **imm**; if it was a mutable object, then for the same reasons as case (2), we can promote it to **capsule**, which is a subtype of **imm**. Note that in all the three cases, a method working on lower security data can be transparently lifted to work on higher security data.

## 4 TYPING RULES

The typing rules are presented in Fig. 4. We assume a reduction similar to Featherweight Java [Igarashi et al. 2001; Pierce 2002]. We have a typing context  $\Gamma ::= x_1 : T_1 \dots x_n : T_n$  which assigns types  $T_i$  to variables  $x_i$ .

**SUB and SUBSUMPTION.** We allow traditional subsumption for modifiers and class names.

However, we are invariant on the security level. We assume our interfaces to induce the standard subtyping between class names.

**T-VAR.** A variable  $x$  is typed using the context  $\Gamma$ .

**FIELD ACCESS.** The result of the field access has the class of the field  $f$ . The security level is the least upper bound of the security levels of  $e_0$  and  $f$ . The resulting modifier is the sum of the modifiers of  $e_0$  and  $f$  as defined in Fig. 2. In this way, if we read a **low mut** Person field from a **high read** receiver, we obtain a **high read** Person result.

**FIELD ASSIGN.** The reference resulting from  $e_0$  has to be **mut** to allow the assignment. The security level of the assigned expression  $e_1$  is the least upper bound of the security levels of expression  $e_0$  and the field  $f$  as declared in  $C$ . For example, if we assign a **high** expression  $e_1$ , either the field  $f$  or the reference resulting from  $e_0$  need to be **high**.

**CALL.** We allow a method call if there is a method type where all parameters and the return value are typable. The security levels of the return type and all **mut** or **capsule** parameters have to be greater than or equal to the security level of the receiver. This requirement is

<sup>4</sup><https://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html>

$$\begin{array}{c}
\frac{C \leq C' \quad mdf \leq mdf'}{s \text{ mdf } C \leq s \text{ mdf}' C'} \text{ (SUB)} \quad \frac{\Gamma \vdash e : T' \quad T' \leq T}{\Gamma \vdash e : T} \text{ (SUBSUMPTION)} \quad \frac{}{\Gamma \vdash x : \Gamma(x)} \text{ (T-VAR)} \\
\\
\frac{\Gamma \vdash e_0 : s_0 \text{ mdf}_0 C_0 \quad s_1 \text{ mdf}_1 C_1 \quad f \in \text{fields}(C_0)}{\Gamma \vdash e_0.f : \text{lub}(s_0, s_1) \text{ mdf}_0 \triangleright \text{ mdf}_1 C_1} \text{ (FIELD ACCESS)} \\
\\
\frac{\Gamma \vdash e_0 : s_0 \text{ mut } C_0 \quad \Gamma \vdash e_1 : \text{lub}(s_0, s) \text{ mdf } C \quad s \text{ mdf } C \quad f \in \text{fields}(C_0)}{\Gamma \vdash e_0.f = e_1 : \text{lub}(s_0, s) \text{ mdf } C} \text{ (FIELD ASSIGN)} \\
\\
\frac{\Gamma \vdash e_0 : T_0 \dots \Gamma \vdash e_n : T_n \quad \text{sec}(T) \geq \text{sec}(T_0) \quad \text{if } mdf(T_i) \in \{\text{mut}, \text{capsule}\} \text{ then } \text{sec}(T_i) \geq \text{sec}(T_0) \quad T_0 \dots T_n \rightarrow T \in \text{methTypes}(\text{class}(T_0), m)}{\Gamma \vdash e_0.m(e_1 \dots e_n) : T} \text{ (CALL)} \\
\\
\frac{\Gamma \vdash e_1 : T_1[s] \dots \Gamma \vdash e_n : T_n[s] \quad \text{fields}(C) = T_1 f_1 \dots T_n f_n}{\Gamma \vdash \text{new } s C(e_1 \dots e_n) : s \text{ mut } C} \text{ (NEW)} \quad \frac{\Gamma[\text{mut} \setminus \text{read}] \vdash e : s \text{ mut } C}{\Gamma \vdash e : s \text{ capsule } C} \text{ (PROM)} \\
\\
\frac{s' \leq s \quad \Gamma \vdash e : s' \text{ mdf } C \quad mdf \in \{\text{imm}, \text{capsule}\}}{\Gamma \vdash e : s \text{ mdf } C} \text{ (SEC-PROM)} \\
\\
\frac{\text{this} : s \text{ mdf } C, x_1 : T_1 \dots x_n : T_n \quad \vdash e : T}{C \vdash s \text{ mdf method } T m(T_1 x_1 \dots T_n x_n) \{\text{return } e; \}} \text{ (M-OK)} \\
\\
\frac{C \vdash M_1 \dots C \vdash M_n \quad mhs(\bar{C}) \subseteq mhs(M_1 \dots M_n)}{\text{class } C \text{ implements } \bar{C} \{ \bar{F} M_1 \dots M_n \}} \text{ (C-OK)} \quad \frac{mhs(\bar{C}) \subseteq \overline{MH}}{\text{interface } C \text{ extends } \bar{C} \{ \overline{MH} \}} \text{ (I-OK)}
\end{array}$$

Fig. 4. Expression typing rules

needed because through dynamic dispatch the receiver may leak information. This is one of the crucial points of our formalism, as explained in Section 2.2.

**NEW.** The newly allocated object is created as a mutable object, and with a specified security level  $s$ . This rule checks that the parameter list  $e_1 \dots e_n$  has the same length as the declared fields. The object of class  $C$  has a list of fields  $f_1 \dots f_n$ . Each parameter  $e_i$  is assigned to a field  $f_i$ . This assignment is allowed if the type of parameter  $e_i$  is (a subtype of)  $T_i[s]$ . The programmer can choose  $s$  to raise the expected security level over the level originally declared for the fields. In order to use an actual parameter with a higher security level ( $s$ ) to initialize a field defined with a lower security level, the newly created object needs to have this chosen security level  $s$ . By using rule SEC-PROM, we can do the opposite, initializing higher security fields with lower security **imm/capsule** values. For example if we have a class `Ex {low Object a; high Object b; topSecret Object c;}`, we can create correct objects with

```

1 new low Ex(lowValue, highValue, topSecretValue)
2 new high Ex(highValue, highValue, topSecretValue)
3 new topSecret Ex(topSecretValue, topSecretValue, topSecretValue)

```

An object `new high Ex(highValue, topSecretValue, topSecretValue)` is incorrect because the second parameter is **topSecret**. Accessing a **high** reference and a **high** field would return a **high** value. This leaks the `topSecretValue` object.

The object `new topSecret Ex(highValue, topSecretValue, topSecretValue)` is only correct if the

$$\frac{\Gamma \vdash e : s \text{ imm Bool} \quad \Gamma[\text{mut}(s), \text{final}(s)] \vdash e_1 : T \quad \Gamma[\text{mut}(s), \text{final}(s)] \vdash e_2 : T}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T} \quad (\text{If})$$

$$\frac{\Gamma \vdash e : s \text{ mdf } C \quad \text{mdf} \in \{\text{imm}, \text{capsule}\}}{\Gamma \vdash \text{declassify}(e) : \perp \text{ mdf } C} \quad (\text{DECL})$$

Fig. 5. Extension: expression typing rules for if and declassify

highValue object can be promoted to **topSecret**. This is only possible for immutable or encapsulated objects.

**PROM.** *Promotion* from **mut** to **capsule** is possible if all **mut** references are only seen as **read** in the typing context. Since a **read** cannot be saved into a field of a **mut** object, we know that the reachable object graph from those **read** variables will not be part of the reachable object graph of the result.

**SEC-PROM.** *Security promotion* raises the security level of a **capsule** or **imm** expression. This captures the intuitive idea that a higher security level is allowed to see all data with lower security levels. However, this is sound only for **capsule** or **imm** expression. An immutable object cannot be modified, so the promotion is secure because no new confidential information can be injected into its ROG. Also a **capsule** object can be passed to a new reference with a higher security level. A leak through the lower **capsule** reference cannot happen, because a **capsule** reference can be used at most one time. Instead, in the case of a mutable object, this assignment would cause a possible leak: it would allow a **high** and **low** alias reference to the same object; and if the **high** reference was updated with **high** data, the **low** reference would see such data as well. Also **read** cannot be promoted: a promoted **read** object can have a **low mut** alias reference to the same object that could now sneakily update the data seen as **high**.

**M-OK.** This rule checks that the definition of a method is well typed. Using the receiver type and the parameter types,  $e$  must have the same type as the declared return type.

**C-OK.** This rule checks that the definition of a class is well typed. The rule uses a helper function  $mhs$  which returns the method headers declared in a set of classes or interfaces, or it directly returns the headers of a set of methods. A well typed class  $C$  implements all methods that are declared in the interfaces  $\bar{C}$ .

**I-OK.** A correct interface must contain all method headers of the implemented interfaces  $\bar{C}$ .

*Implicit Information Flows.* The language as presented is minimal but using well-known encodings it can support imperative update of local variables (use box objects with a single field and field updates) and conditionals (use any variation of the Smalltalk [Goldberg and Robson 1983] way to support control structures). However, in Fig. 5, for the sake of a more compelling explanation, we show how the **if** construct could be typed if we expand our language with **if**, Booleans and updatable local variables.

**IF.** For a conditional statement with a **Bool** expression in the guard, we define that both branches  $e_1$  and  $e_2$  must have the same type  $T$ . With  $\Gamma[\text{mut}(s), \text{final}(s)]$ , we introduce two restrictions: with  $\text{mut}(s)$ , we prevent mutation of **mut** objects with a security level lower than  $s$  by seeing them as **read**, and with  $\text{final}(s)$ , we prevent updating all local variables with a security level lower than  $s$ . If  $s$  is the lowest security level, both functions do not restrict  $\Gamma$ .

Therefore, assignments to less confidential variables or fields in the branches are prohibited to prevent leaks. This means that if the expression in the guard of a conditional statement has

```

1  interface I {
2    low mut method low imm I doIt(low imm I l1, low imm I l2);
3  }
4  class C1 implements I {
5    low mut method low imm I doIt(low imm I l1, low imm I l2) {
6      return l1; }
7  }
8  class C2 implements I {
9    low mut method low imm I doIt(low imm I l1, low imm I l2) {
10     return l2; }
11 }
12 ...
13 high mut C1 c1 = new high C1();
14 high mut C2 c2 = new high C2();
15 low imm C1 l1 = new low C1();
16 low imm C2 l2 = new low C2();
17 high mut I i = c1; //c1 assigned to reference of type I
18 low imm I x = i.doIt(l1,l2); //ILL TYPED in our system
19 //if it was accepted, by observing low variable x= l1
20 //we could deduce the content of the high variable i = c1
21
22 //equivalent behaviour using an if
23 if (i instanceof C1){ x = l1; } else { x = l2; }

```

Listing 3. Ill-typed example of a method call

a security level that is higher than the lowest security level, only assignments to variables of at least the security level of the guard are allowed. Additionally, only mutable objects of at least the security level of the guard can be mutated. In this way, only data whose security level is at least the one of the guard can be mutated.

Using the Smalltalk-style as discussed above, our pre-existing rules would handle the encoded code exactly as with the explicit IF-rule, Booleans, and local variables. Thus, our system is minimal, but does not lack expressiveness. The IF-rule has a similar constraint as the CALL-rule where return type and **mut** and **capsule** parameters have to be greater than or equal to the security level of the receiver.

The following example, in both OO style and with an explicit **if** shows the mechanism to prevent implicit information flow leaks: In Listing 3, we have an interface **I** that declares a method **doIt** that gets two **low imm I** parameters as input. The classes **C1** and **C2** implement the interface and the method. The implemented method of **C1** returns the first parameter and the implemented method of **C2** returns the second parameter. We initialize two **high** variables **C1 c1** and **C2 c2** and two **low** variables **C1 l1** and **C2 l2**. By assigning **c1** to **i** in Line 17, we hide the information of the explicit class behind the interface. However, if we are allowed to execute Line 18, calling the method **doIt**, the class **c1** is revealed because **l1** is returned. The attacker gets the information about the explicit class by observing the return value. This example is an object-oriented implementation of a conditional statement. If we use an **if** instead of dynamic dispatch, the leak is clearly visible (see Line 23), as we assign **low** variables in branches of a **high** guard. To prevent such leaks, we define that the security level of the return type is never lower than the security level of the receiver. Information does not only flow through the method result: also **mut** or **capsule** parameters can be used to push information out of the method; thus also the security level of **mut** and **capsule** parameters must never

be lower than the security level of the receiver. Thus, in both cases with the CALL-rule and with the IF-rule the implicit leak is prevented.

*Declassification.* A **mut** or **read** reference can only be assigned to references of the same security level. However, since **imm** and **capsule** references are referentially transparent, it is safe to assign an **imm** or **capsule** object from a lower security level to a higher one. On the other hand, we are not allowed to assign an object with a higher security level to a reference with a lower security level. Similar to the **if**, we consider adding a **declassify** operator that can be used to manipulate the security level  $s$  of expressions to allow a reverse information flow in appropriate cases. In some cases, this reverse information flow is needed to develop meaningful programs. For example, if a confidential password is hashed, the value should be assignable to a public output. With the **declassify** expression, the security level of a **capsule** or **imm** reference is set to the lowest security level. In Fig. 5, the DECL rule is shown.

**DECL.** The **declassify** rule is used to change the security level of **capsule** or **imm** expressions to the bottom level of the lattice.

**declassify** should be used with caution because secure information is leaked in the case of inappropriate use. In this rule, we cannot declassify an expression to a specific security level, but this is not a limitation, since we can encapsulate declassification statements inside of methods which directly promote the declassified expression to the desired security level. **mut** and **read** references cannot be declassified because potentially existing aliases are a security hazard. For example, if you declassify a **high mut** reference and a **high read** alias still exists, an attacker could use that now **low mut** reference to mutate information visible as **high**. Declassification is not part of our system to type check secure programs, and we do not need it to make secure programs, rather it is a mechanism to break security in a controlled way. That is, when comparing with examples of other papers [Myers 1999], we do not use **declassify** to encode behavior. We can still use it to print out results to show that the code is working. In the DECL rule that we present, **declassify** is just a special expression. In the full SIFO language embedded in L42, declassification can be flexibly tuned to the user needs preventing accidental declassification.

## 5 PROOF OF NONINTERFERENCE

In this section, we aim to ensure *noninterference* [Goguen and Meseguer 1982] according to our information flow policy. Noninterference is a central criterion for secure information flow, as we want to ensure that an attacker cannot deduce confidential data by observing data with lower security levels. It is based on the indistinguishability of program states. Two program states are indistinguishable (also referred to as *observably similar*) up to a certain security level if they agree on their memory reachable from references with a security level lower than that specific security level. Using this property, a program satisfies the noninterference Theorem 5.0 if and only if the following holds: if a program is executed in two observably similar memories up to a certain security level, then the resulting memories are also observably similar up to the same security level, but may differ in higher security levels.

**THEOREM 5.0 (GENERAL NONINTERFERENCE).**

*If we have expressions  $e_1$  and  $e_2$  without declassification that are well typed and have the same low values, but possible different high values ( $e_1$  lowEqual  $e_2$  see Definition 5.6),  $M_1$  and  $M_2$  are well typed memories,  $M_1$  and  $M_2$  are low observably similar,  $M_1|e_1 \rightarrow^* M'_1|v_1$ ,  $M_2|e_2 \rightarrow^* M'_2|v_2$ , then  $M'_1$  and  $M'_2$  are low observably similar, and memories  $M'_1$ ,  $M'_2$ , values  $v_1$ , and  $v_2$  are well typed and  $v_1$  lowEqual  $v_2$ .*

In this section, we prove noninterference for a lattice with a **low** and a **high** security level (**low** ≤ **high**) for terminating programs. Nonterminating programs and programs with an arbitrary lattice

$$\begin{aligned}
e &::= x \mid e.f \mid e.f = e \mid e.m(\bar{e}) \mid \mathbf{new} \ s \ C(\bar{e}) \mid v \\
\mathcal{E}_v &::= [] \mid \mathcal{E}_v.f \mid \mathcal{E}_v.f = e \mid v.f = \mathcal{E}_v \mid \mathcal{E}_v.m(\bar{e}) \mid v.m(\bar{v} \ \mathcal{E}_v \ \bar{e}) \mid \mathbf{new} \ s \ C(\bar{v} \ \mathcal{E}_v \ \bar{e}) \\
\mathcal{E} &::= [] \mid \mathcal{E}.f \mid \mathcal{E}.f = e \mid e.f = \mathcal{E} \mid \mathcal{E}.m(\bar{e}) \mid e.m(\bar{e} \ \mathcal{E} \ \bar{e}') \mid \mathbf{new} \ s \ C(\bar{e} \ \mathcal{E} \ \bar{e}') \\
v &::= s \ \mathbf{mdf} \ o \\
M &::= o_1 \mapsto C_1(\bar{o}_1) \dots o_n \mapsto C_n(\bar{o}_n)
\end{aligned}$$

Fig. 6. Runtime syntax and values

$$\begin{aligned}
&\frac{M|e \rightarrow M'|e'}{M|\mathcal{E}_v[e] \rightarrow M'|\mathcal{E}_v[e']} \quad (\text{CTX}) && \frac{v = s \ \mathbf{mdf} \ o \quad o \mapsto C(o_1 \dots o_n) \in M \quad \mathit{fields}(C) = T_1 \ f_1 \dots T_n \ f_n}{M|v.f_i \rightarrow M|\mathit{lub}(s, \mathit{sec}(T_i)) \ \mathbf{mdf} \triangleright \ \mathbf{mdf}(T_i) \ o_i} \quad (\text{FIELD ACCESS}) \\
&\frac{\mathit{fields}(C) = \_ T_0 \ f_0 \dots T_n \ f_n \quad v_1 = s \ \mathbf{mdf} \ o \quad v_2 = \_ \ o' \quad v'_2 = \mathit{lub}(s, \mathit{sec}(T_0)) \ \mathbf{mdf}(T_0) \ o'}{M, o \mapsto C(\bar{o} \ o_0 \dots o_n) | v_1.f_0 = v_2 \rightarrow M, o \mapsto C(\bar{o} \ o' \ o_1 \dots o_n) | v'_2} \quad (\text{FIELD UPDATE}) \\
&\frac{T'_i = s'_i \ \mathbf{mdf}'_i \ C_i \quad v_i = s_i \ \mathbf{mdf}_i \ o_i \quad v'_i = s'_i \ \mathbf{mdf}'_i \ o_i \quad o_0 \mapsto C_0(\bar{o}) \in M \quad p(C_0) = \mathbf{class} \ C_0 \{ \_ s \ \mathbf{mdf} \ \{ \mathbf{method} \ T \ m(T_1 \ x_1 \dots T_n \ x_n) \{ \mathbf{return} \ e \} \_ \} \quad T'_0 \dots T'_n \rightarrow T' = \mathit{mostSpecMethType}(C_0, m, s_0 \dots s_n, \mathbf{mdf}_0 \dots \mathbf{mdf}_n)}{M|v_0.m(v_1 \dots v_n) \rightarrow M|e[\mathit{this} \setminus v'_0, x_1 \setminus v'_1, \dots, x_n \setminus v'_n]} \quad (\text{CALL}) \\
&\frac{v_1 = s_1 \ \mathbf{mdf}_1 \ o_1 \dots v_n = s_n \ \mathbf{mdf}_n \ o_n}{M|\mathbf{new} \ s \ C(v_1 \dots v_n) \rightarrow M, o \mapsto C(o_1 \dots o_n) | s \ \mathbf{mut} \ o} \quad (\text{NEW})
\end{aligned}$$

Fig. 7. Reduction rules

do typecheck, and we expect noninterference to work for those too, but our proof technique does not address those cases. We also do not include the **declassify** operation because this rule explicitly allows **high** data to interfere with data of lower security levels. This means, **declassify** is only an explicit mechanism to break security in a controlled way, as we explained in Section 4. In the section, we use the notation of memory and expression. The meaning of these terms are overloaded. A memory is often a stack in a while languages, in object-oriented languages, the memory is often a heap. In our expression-based language, we model a heap (a map from memory locations to the values stored for each field as defined by the location's class), and we use the expressions themselves to track the values that are accessible within each expression (similar to how the stack achieves this during the execution). This means, memory  $M$  captures the heap and expression  $e$  captures the stack and any inputs. We cannot model the whole memory without the expression.

## 5.1 Reduction Rules

SIFO is an additional type system layer and does not influence the language semantics. However, for the sake of the noninterference proof, we need to instrument the small-step reduction to keep track of security and modifiers during program execution. To this aim, we define in Fig. 6 values  $v$  as a location  $o$  in a store with security level and type modifier. The store is some memory  $M$ . In  $M$ , a location points to some class  $C$  where each field is again a location  $o_i$  in the memory  $M$ . With the evaluation context  $\mathcal{E}_v$ , we define the order of evaluation. We assume two well-formedness properties. The memory is well-formed if  $M$  is a map from  $o$  to  $C(\bar{o})$ , thus the locations  $\bar{o}$  in domain of  $M$  are unique. The reduction arrow  $(M|e \rightarrow M'|e')$  is well-formed if  $M$  and  $M'$  have no dangling pointers with respect to  $e$  and  $e'$  (i.e., every pointer points to a valid object in the memory). In Figure 7, the following reduction rules are shown.



*Definition 5.1 (mostSpecMethType).*

- $mostSpecMethType(C, m, ss, mdfs) = openCapsules(Ts' \rightarrow T, mdfs)$   
 $s \text{ mdf method } T_0 \ m(T_1 \ x_1 \ \dots \ T_n \ x_n) \ \text{in } C$   
 $raiseFormalSecurity(s \ mdf \ C \ T_1 \ \dots \ T_n \ \rightarrow \ T_0, ss) = Ts \ \rightarrow \ T$   
 $raiseActualSecurity(Ts, ss, mdfs) = Ts'$
- $raiseFormalSecurity(T_1 \ \dots \ T_n \ \rightarrow \ T_0, s'_1 \ \dots \ s'_n) = T'_1 \ \dots \ T'_n \ \rightarrow \ T'_0$   
 $T_i = s_i \ \text{mdf}_i \ C_i$   
 $T'_i = lub(s, s_i) \ \text{mdf}_i \ C_i$   
 $s = lub(\{s'_i \mid s'_i > s_i\})$
- $raiseActualSecurity(T_0 \ \dots \ T_n, s_0 \ \dots \ s_n, mdf_0 \ \dots \ mdf_n) = T'_0 \ \dots \ T'_n$   
 $T_i = s'_i \ \text{mdf}_i \ C_i$   
 $T'_i = sec(T_i) \ \text{mdf}_i \ C_i$   
*if*  $s_i < sec(T_i)$  *then*  $mdf_i \in \{\text{capsule, imm}\}$
- $openCapsules(T_1 \ \dots \ T_n \ \rightarrow \ T, mdf'_1 \ \dots \ mdf'_n) = T'_1 \ \dots \ T'_n \ \rightarrow \ T$   
 $T_i = s_i \ \text{mdf}_i \ C_i$   
 $T'_i = s_i \ \text{mdf}'_i \triangleright \text{mdf} \ C_i$

$$\frac{M \mid (\text{high mdf } o).m(\bar{v}) \rightarrow^+ M' \mid (s \ \text{mdf} \ o')}{M \mid \mathcal{E}_v[(\text{high mdf } o).m(\bar{v})] \rightarrow_{call} M' \mid \mathcal{E}_v[(\text{high mdf } o')]} \quad (\text{HREC})$$

$$\frac{e \text{ not of form } \mathcal{E}_v[(\text{high mdf } o).m(\bar{v})] \quad M \mid e \rightarrow M' \mid e'}{M \mid e \rightarrow_{call} M' \mid e'} \quad (\text{HOTHER})$$

Fig. 8. Additional reduction rules for the noninterference proof

**CTX.** This is the conventional contextual rule, allowing the execution of subexpressions.

**FIELD ACCESS.** A field access  $f_i$  of a value  $v$  is reduced to a location  $o_i$  if the location  $o$  of  $v$  points to the suitable class  $C(o_1 \ \dots \ o_n)$ . The security level is the least upper bound of the security levels of  $v$  and  $f_i$  and the modifier is the sum of modifiers of  $v$  and  $f_i$ .

**FIELD UPDATE.** The store  $o \mapsto C(\bar{o} \ o_0 \ \dots \ o_n)$  is updated with an assignment of  $v_2$  to the field  $f_0$ . The location  $o_0$  is replaced with  $o'$ . The security level of the resulting value  $v'_2$  is the least upper bound of the security levels of expression  $v_0$  and the field  $f_0$  as declared in  $C$ . The type modifier of  $v'_2$  is equal to the type modifier of the field  $f_0$  as declared in  $C$ .

**CALL.** We reduce a method call to an expression  $e$ , where each value  $v_i$  is assigned to a parameter  $x_i$ . As we use multiple method types, the actual assigned values  $v'_i$  can have an updated security level or type modifier. Additionally, the called method has to be declared in the class  $C_0$  pointed to by the location in  $v_0$ . The concrete calculation of the updated types  $T'_i$  with *mostSpecMethType* is shown in Definition 5.1. The definition calculates the exact



method types to support the proof of noninterference. To calculate the types  $T'_i$ , the functions needs as parameters, the class name  $C$ , the method name  $m$ , and the used security levels  $ss$  and type modifiers  $mdfs$  to call this method. First with *raiseFormalSecurity*, for each formal parameter type  $T_i$ , the security level can be raised. We collect all security levels  $s'_i$  where a security level higher than declared in the formal parameter is passed. The least upper bound of the collected security levels is the minimum security level for all actual parameters. It is allowed that formal parameter have higher security levels than  $s$  if a current parameter is passed with the same security level higher than  $s$ . With *raiseActualSecurity*, it is checked that the actual parameters have the same security level as the now raised formal parameters. Only actual parameters with type modifier `imm` or `capsule` can be raised to the needed level. In the last step, the combined type modifier of the actual and formal parameter is calculated with the function of Fig. 2.

**NEW.** The newly created object is reduced to a location  $o$  in the memory that points to the class  $C$  where each field is again a location  $o_i$ . The reduced value has the same security level  $s$  as in the expression before the reduction and a `mut` type modifier. Note that we do not need to say that a new reference is not in the domain of the old memory. This is implicit by the well-formedness of the memory.

**HREC, HOTHER.** The two rules in Fig. 8 are added to condense the reduction of methods calls on high receivers ( $\rightarrow_{call}$ ). These rules only consider a `low`  $\leq$  `high` lattice of security levels and are needed for our noninterference proof for technical reasons. We will prove noninterference only for a `low`  $\leq$  `high` lattice, but the typing rules work with any security lattice. As you can see, we condense all execution steps that happen under the control of a `high` receiver into a single more abstract step. Here,  $\rightarrow^+$  is the transitive closure. Of course, rule (HREC) is not applicable when the reduction of such a method does not terminate; this is why our proof technique only works on terminating programs: it does not make sense to talk about noninterference for a program stuck into a non-terminating loop inside of a `high` method. This program would never reach again a state where a `low` attacker may attempt to observe data.

## 5.2 Definition of Similarity w.r.t. Security Levels

In this subsection, we define the (observable) similarity of two memories for the proof of noninterference. Therefore, we need further definitions of reachable object graphs (ROG). We define mutable, `low`, and `high` memories using a notation of reachable object graph. Definition 5.2 defines the mutable ROG of a memory and the current state of the main expression  $e$ . Note how this is similar to ownership work where the stack is used to track the current top level state. In an expression-based language, the stack is represented as the set of values inside of the main expression. The MROG contains every location inside of the expression  $e$  that has a `mut` or a `capsule` type modifier. Additionally, `mut` fields of locations in the MROG are included. In Definition 5.3, the `low` ROG contains every location inside of the expression  $e$  that has a `low` security level. Additionally, `low` fields of locations in the `low` ROG are included.

*Definition 5.2.*  $MRog(M, e) = \bar{o}$

- $o \in MRog(M, e)$  if:  
 $e = \mathcal{E}[s \text{ mut } o]$
- $o \in MRog(M, e)$  if:  
 $e = \mathcal{E}[s \text{ capsule } o]$
- $o_i \in MRog(M, e)$  if:  
 $o \in MRog(M, e)$   
 $o \mapsto C(o_1 \dots o_n)$  in  $M$   
`class C _ {T1f1 ... Tnfn_}`  
 $mdf(T_i) = \text{mut}$

*Definition 5.3.*  $lowRog(M, e) = \bar{o}$

- $o \in lowRog(M, e)$  if:  
 $e = \mathcal{E}[\text{low\_} o]$
- $o_i \in lowRog(M, e)$  if:  
 $o \in lowRog(M, e)$   
 $o \mapsto C(o_1 \dots o_n)$  in  $M$   
`class C _ {T1f1 ... Tnfn_}`  
 $sec(T_i) = \text{low}$

In Definition 5.4, the **high** ROG includes locations of **high mut** values. The **high** ROG also includes every **mut** value that is pointed to by a location in the **high** ROG. Furthermore, the **high mut** fields of a location in the **low** ROG are included. We exclude **high imm** values, as **imm** values can be referenced by both **low** and **high** references.

*Definition 5.4.*  $highRog(M, e) = \bar{o}$

- $o \in highRog(M, e)$  if :  
 $e = \mathcal{E}[\text{high mut } o]$
- $o_i \in highRog(M, e)$  if :  
 $o' \in highRog(M, e)$   
 $o' \mapsto C(o_1 \dots o_n)$  in  $M$   
`class C _ {T1f1 ... Tnfn_}`  
 $mdf(T_i) = \text{mut}$
- $o_i \in highRog(M, e)$  if :  
 $o' \in lowRog(M, e)$   
 $o' \mapsto C(o_1 \dots o_n)$  in  $M$   
`class C _ {T1f1 ... Tnfn_}`  
 $sec(T_i) = \text{high}$   
 $mdf(T_i) = \text{mut}$   
 $o_i \in \text{dom}(M)$

In Definition 5.5, we define the observable similarity of two memories. Two memories  $M_1$  and  $M_2$  are similar given an expression  $e$ , if and only if the **low** locations of both memories are equal. As explained before, the expression  $e$  is needed to track the top level state. We need a transformation on the memories to filter **high** locations and **high** fields of classes pointed by **low** locations. As we are only interested in similarity of **low** locations for the noninterference proof, these **high** locations have to be filtered. The filtering is defined by  $M[\text{only } \bar{o}]$ . In the given memory  $M$ , each location  $o$  is removed that is not in the input set  $\bar{o}$ . Additionally, for the  $o$  in the input set  $\bar{o}$ , each  $o_i$  in  $o \mapsto C(o_1 \dots o_n)$ , where the corresponding field in class  $C$  is defined with a **high** security level, is replaced with the location  $o$  to filter the explicit high location  $o_i$ . Thus, with this transformation two memories are equal, if they differ only in the **high** ROG.

Removing the **high** locations in this way may produce an ill type memory; this is not a problem since those memories are only used as a device to define  $\_similar(e)\_$  and not in the reduction.

*Definition 5.5.*

$M_1 \text{ similar}(e) M_2 \leftrightarrow M_1[\text{only } lowRog(M_1, e)] = M_2[\text{only } lowRog(M_2, e)]$

where  $M[\text{only } \bar{o}] = M'$  is defined as:

- $(o_1 \mapsto C_1(\bar{o}_1) \dots o_n \mapsto C_n(\bar{o}_n))[\text{only } \bar{o}] =$   
 $o_1 \mapsto C_1(\bar{o}_1)[\text{only } \bar{o}] \dots o_n \mapsto C_n(\bar{o}_n)[\text{only } \bar{o}]$
- $(o \mapsto C(\_))[\text{only } \bar{o}] = \text{empty}$  if  $o$  is not in  $\bar{o}$
- $(o \mapsto C(o_1 \dots o_n))[\text{only } \bar{o}] = o \mapsto C(o'_1 \dots o'_n)$  if  $o$  in  $\bar{o}$  with:  
 $fields(C) = T_1 f_1 \dots T_n f_n$   
 $o'_i = o_i$  if  $sec(T_i) = \text{low}$   
 $o'_i = o$  if  $sec(T_i) = \text{high}$

The Definition 5.6 compares that two expressions are equal if we only consider the **low** values. It is defined as rule induction, where the only interesting case is that **high** values are ignored. This means, two expressions are *lowEqual* if either they are the same **low** expression, or possibly-different **high** locations.

*Definition 5.6.*  $e$  *lowEqual*  $e'$

- $x$  *lowEqual*  $x$
- $e.f$  *lowEqual*  $e'.f$  iff  $e$  *lowEqual*  $e'$
- $e_0.f = e'_0.f$  *lowEqual*  $e_1.f = e'_1.f$  iff  $e_0$  *lowEqual*  $e_1$  and  $e'_0$  *lowEqual*  $e'_1$
- $e_0.m(e_1 \dots e_n)$  *lowEqual*  $e'_0.m(e'_1 \dots e'_n)$  iff  $e_i$  *lowEqual*  $e'_i$  for  $i$  in  $0 \dots n$
- $\text{new } s \ C(e_1 \dots e_n)$  *lowEqual*  $\text{new } s \ C(e'_1 \dots e'_n)$  iff  $e_i$  *lowEqual*  $e'_i$  for  $i$  in  $1 \dots n$
- $(\text{low mdf } o)$  *lowEqual*  $(\text{low mdf } o)$
- $(\text{high mdf } o)$  *lowEqual*  $(\text{high mdf } o')$

The Definition 5.7 is essential to constrain reduction: an alternative to our reduction that is undesirable could trivially preserve security by adding everything to the **high** memory so that no **low** objects remain. The definition of *preserves* constrains the reduction to only change the **high** memory in the few appropriate and necessary cases as follows: In the first trivial case, nothing is added. In the second case, a new **high** object is created and added to the **high** ROG. In the third case, a **low capsule** object is promoted to **high** and added to the **high** ROG. Here, all options of a promotion of a **capsule** object are shown (field assign, method call as receiver and as parameter, and object construction as a parameter).

*Definition 5.7.*  $M$  *preserves*( $e/e'$ )  $M'$  if one of the three holds:

- 1)  $\text{highRog}(M, e) = \text{highRog}(M', e')$
- 2)  $\text{highRog}(M, e), o = \text{highRog}(M', e')$  then  
 $e = \text{ctx}_0[\text{new high } C(\bar{v})], e' = \text{ctx}_0[\text{high mdf } o]$
- 3)  $\text{highRog}(M, e), \text{MROG}(M, \text{high mdf } o) = \text{highRog}(M', e')$  then  
 $e = \text{ctx}_0[\text{low capsule } o], e' = \text{ctx}_1[\text{high mdf } o]$   
 and  $e$  is equal to one of the following:  
 $\text{ctx}[v.f = \text{low capsule } o]$   
 $\text{ctx}[(\text{low capsule } o).m(\bar{v})],$   
 $\text{ctx}[v.m(\bar{v}(\text{low capsule } o)\bar{v}'),]$   
 $\text{ctx}[\text{new } C(\bar{v}(\text{low capsule } o)\bar{v}')] ]$

### 5.3 Noninterference Theorem and Proof

We prove noninterference according to our information flow policy. In literature, there are many proposed languages (with proofs) that are very similar to the type system proposed in this work [Giannini et al. 2019]. Here, to avoid repeating those same proofs that are already presented in those other works, we accept two assumptions. (1) Soundness: the reduction does not get stuck. (2) Immutability and encapsulation: In addition to not getting stuck, the reduction also never mutates the ROG of an immutable object, and the ROG of capsules is always encapsulated (i.e., all mutable objects can be reached only through the **capsule** reference). Both assumptions are established and proved before [Giannini et al. 2019].

To prove noninterference, we first introduce two lemmas that facilitate the proof. We show that the reduction terminates using  $\rightarrow_{\text{call}}$  and we show that, given two similar memories, each reduction step results in similar memories.

*Call-Reduction Termination.* We prove in Lemma 5.8 that the reduction  $\rightarrow_{\text{call}}$  does not interfere with termination: if we have a well typed memory  $M$  and an expression  $e$  and the program terminates

with the normal reduction, then it also terminates with  $\rightarrow_{call}$ . The result for both reductions is also the same.

LEMMA 5.8 ( $\rightarrow_{call}$  TERMINATION). *If memory  $M$  and expression  $e$  are well typed and if the reduction terminates with  $\rightarrow$ , then it terminates also with the reduction  $\rightarrow_{call}$  with the same result.*

PROOF. This can be verified by cases:  $\rightarrow_{call}$  behaves exactly like  $\rightarrow$ , but has a different granularity of the steps. Therefore,  $\rightarrow_{call}$  terminates in every case where  $\rightarrow$  terminates.  $\square$

*Bisimulation.* To establish noninterference, Lemma 5.9, the bisimulation core, states that two well typed and similar memories  $M_1$  and  $M_2$  and expressions  $e_1$  and  $e_2$ , where  $e_1$  *lowEqual*  $e_2$  holds, reduce to  $M'_1|e'_1$  and  $M'_2|e'_2$  that are also similar, and the reduced expressions  $e'_1$  and  $e'_2$  are *lowEqual*. We need property (2) to state that both memories are observably similar, and property (3) that also the expressions are similar regarding the observable values. Both properties together represent observably similar memories as in Theorem 5.0. Furthermore, the preservation property of each memory ensures that the reduction only changes the **high** memory in necessary cases. With this lemma, we know that each reduction step from similar memories results in similar memories: each  $\rightarrow_{call}$  reduction step ensures noninterference.

LEMMA 5.9 (BISIMULATION CORE).

*Given well typed memories and expressions without declassification  $M_1, M_2, e_1$ , and  $e_2$  where  $M_1|e_1 \rightarrow^* \_ |v_1$  and  $M_2|e_2 \rightarrow^* \_ |v_2$ .*

*If the following holds*

- 1)  $M_1|e_1 \rightarrow_{call} M'_1|e'_1$ ,
- 2)  $M_1$  *similar*( $e_1$ )  $M_2$ ,
- 3) and  $e_1$  *lowEqual*  $e_2$ ,

*then:*

- A)  $M_2|e_2 \rightarrow_{call} M'_2|e'_2$  and  $e'_1$  *lowEqual*  $e'_2$ ,
- B)  $M'_1$  *similar*( $e'_1$ )  $M'_2$ ,
- C)  $M_1$  *preserves*( $e_1/e'_1$ )  $M'_1$ ,
- D)  $M_2$  *preserves*( $e_2/e'_2$ )  $M'_2$ ,
- E)  $M'_1, M'_2, e'_1, e'_2$  are well typed

PROOF. We prove that all five conditions A–E are satisfied. For A and B, we prove this theorem by cases on  $\rightarrow_{call}$ . We only prove the cases including a context, because the proofs with an empty context imply the correctness of the rules without a context.

**Proof of A and B by cases:**

**Case CTX + CALL:**

If  $e_1$  and  $e_2$  are of form  $\mathcal{E}_v[(\mathbf{high} \text{ mdf } o).m(\bar{o})]$ , the proof is by rule (HREC).

*Proof of A:* The only point of non-determinism in this language is the way new object identities are chosen, and the only way to introduce a new  $o$  is with the NEW rule. From assumption (1) and Lemma 5.8, we know that there is an execution of  $\rightarrow_{call}$ , containing an arbitrary number of reduction steps ( $\rightarrow$ ). The reduction is of form  $M|\mathcal{E}_v[(\mathbf{high} \text{ mdf } o).m(\bar{v})] \rightarrow_{call} M'|\mathcal{E}_v[(\mathbf{high} \text{ mdf } o'')]$ , where the result is a **high** value, thus by the definition of *lowEqual*,  $\mathcal{E}_v[(\mathbf{high} \_ \_)]$  *lowEqual*  $\mathcal{E}_v[(\mathbf{high} \_ \_)]$  holds.

*Proof of B:* We execute a number of steps on the **high** receiver. By assumption (1) and Lemma 5.8, this process terminates and produces a result. By typing rule CALL, we know the result is **high** and the parameters  $vs$  must only contain **low read/imm/capsule** values. By the assumption that the language satisfies the modifier properties (e.g. immutability), we do not modify the ROG of the **low read/imm** parameters. The **low** capsules are promoted to **high** in both memories, and thus, will not be

part of the **low** ROG anymore.

All the other parameters and the receivers are **high**, so they will not influence the **low** values: the whole ROG of a **high** object is **high**, and computation can only influence reachable objects since we do not have any static variables. So  $B$  holds since the (shrunk) **low** ROG on both memories cannot be mutated.

If  $e_1$  and  $e_2$  are not of form  $\mathcal{E}_v[(\mathbf{high} \text{ mdf } o).m(\bar{o})]$ , the proof is by rule (HOTHER).

*Proof of A:* In this case we are doing a single reduction step  $\rightarrow$ . The only way to introduce non-determinism is by creating a new object, but this step is a method call. Thus,  $M_1 = M'_1$ ,  $M_2 = M'_2$  and  $e'_1 \text{ lowEqual } e'_2$ .

*Proof of B:* The expressions  $e_1$  and  $e_2$  must be of form  $\mathcal{E}_v[(\mathbf{low} \text{ mdf } o).m(\bar{o})]$  and the **low** receiver is the same on both sides. Moreover, since the original memories are similar, the **low**  $o$  is an instance of the same class, thus the method call will resolve to the same body, and the application of reduction rule (CALL) is deterministic and is identically in both cases, so the memories are not influenced (except for the usual shrinkage on promoted **low capsule** and **low imm** objects). Thus,  $B$  holds.

### Case CTX + FIELD UPDATE:

*Proof of A:* By assumption (1), we know the expression reduces. The only way to introduce a new  $o$  is with the (NEW) rule; thus  $A$  holds since we use the same process to get  $v'_2$  (the rule applies a deterministic procedure).

*Proof of B:* In this case, if  $v_2$  is **low**, we know:  $M_1|\mathcal{E}_v[v_1.f_0 = v_2]$ ,  $M_2|\mathcal{E}_v[v_1.f_0 = v_2]$ , and  $M_1 \text{ similar}(\mathcal{E}_v[v_1.f_0 = v_2]) M_2$ . Thus,  $M'_1|v'_2$ ,  $M'_2|v'_2$ , and  $M'_1 \text{ similar}(\mathcal{E}_v[v'_2]) M'_2$ .

If  $v_2$  is **high**, then we could have a different value in the second reduction, but this value will also be **high**, and thus, not influence similarity. By well typing, this **high** value will be stored in a **high** field. Intuitively, if  $v_1$  is **low**, we can assign a **high** only if  $f_0$  is **high**. In this case, the **low** ROG did not contain  $v_2$  and now does not contain  $v'_2$ , so  $B$  holds. If  $v_1$  is **high**, then it was not part of the **low** ROG to begin with, so  $B$  holds.

However, we need to inspect all possible field update cases to check that all assignments do not violate our similarity property:

To shorten the writing for all cases, we assume a **low** reference  $\text{low1}$ , a **high** reference  $\text{high1}$  with a **low** field  $\text{lowF}$  and a **high** field  $\text{highF}$ . The assigned objects can be **low**  $\text{low2}$  or **high**  $\text{high2}$ .

- $\text{low1}.\text{lowF}=\text{low2}$  This is equal in both reduction, so  $B$  holds.
- $\text{low1}.\text{highF}=\text{low2}$  This is ok if **low** is **imm** or **capsule**. If we have removed the last **low** reference to 'low', then the result of  $\_[\text{only } \_]$  will shrink in the same way in both computations.
- $\text{high1}.\text{highF}=\text{low2}$  This is ok if **low** is **imm** or **capsule**, and thus, it is absent in the **low** ROG.
- $\text{high1}.\text{lowF}=\text{low2}$  This is ok if **low** is **imm** or **capsule**, and thus, it is absent in the **low** ROG.

Note for the **capsule** cases: The value  $v_2$  was **low capsule** in  $\mathcal{E}_v[v_1.f_0 = v_2]$ , thus it was kept as part of the **low** memory by  $\_[\text{only } \_]$ . In the next step  $v'_2$  is now **high**, thus it is not kept as part of the **low** memory by  $\_[\text{only } \_]$ . Since the only change between the result of the  $\_[\text{only } \_]$  memory is the absence of the **low capsule**,  $B$  holds.

- $\text{low1}.\text{lowF}=\text{high2}$  This is forbidden by the type system.
- $\text{low1}.\text{highF}=\text{high2}$  This is ok and irrelevant, since it does not change the **low** ROG.
- $\text{high1}.\text{highF}=\text{high2}$  This is ok and irrelevant, since it does not change the **low** ROG.
- $\text{high1}.\text{lowF}=\text{high2}$  This is ok and irrelevant, since it does not change the **low** ROG.

### Case CTX + FIELD ACCESS:

*Proof of A:* The only way to introduce non-determinism is the way new objects are created. We are accessing a location  $o$  that is equal in both cases or it is a **high** location. As we are not modifying the memories,  $M_1 = M'_1$  and  $M_2 = M'_2$  and  $e'_1 \text{ lowEqual } e'_2$  holds.



*Proof of B:* The memory is not changed by accessing a field.  $M_1 \text{ similar}(e) M_2$  holds before and  $M_1 = M'_1$  and  $M_2 = M'_2$ , thus  $B$  holds.

### Case CTX + NEW:

In this case, assumption (1) is of form  $M_1 | \mathcal{E}_v[\text{new } s \ C(\bar{v})] \rightarrow_{\text{call}} M_1, o \mapsto C(\_) | \mathcal{E}_v[(s \ \text{mut } o)]$ , and (A) is similar with  $M_2$ .

*Proof of A:* This can be obtained by simply choosing a suitable reference ‘o’ for the NEW rule.

*Proof of B:* It holds because the new memories grow by adding the same exact object on both sides.

### Proof of C and D:

To prove the conditions  $C$  and  $D$ , we have to show that if  $M$  and  $e$  are well typed and  $M|e \rightarrow_{\text{call}} M'|e'$  then  $M \text{ preserves}(e/e') M'$ . That means, each reduction step ensures the preserve property that the **high** memory is only changed in necessary cases. This statement holds by the construction of our reduction rules. We only promote expressions to **high** if it is necessary. No unnecessary promotions are done. The only changes from **low** to **high** are explicitly stated in Definition 5.7: creation of a new **high** object and the promotion of a **low capsule** expression. Thus,  $C$  and  $D$  holds.

### Proof of E:

Proving condition  $E$  is more complex. From our assumptions we conclude that the well typedness of the base language is not violated, since the SIFO type system is just stricter than the regular L42 system. However, we need to prove that the  $\rightarrow_{\text{call}}$  reduction preserves the added security typing. This is quite subtle thanks to multiple method types (Fig. 3): The type system as presented does not respect preservation; that is, when calling a method that has been typed using multiple method types, the inlined body of the method may not respect our provided type system. However, we are using the  $\rightarrow_{\text{call}}$  reduction, and the call reduction skips in a single step to the full method evaluation. The method body will evaluate to a value; we have not formally specified typing rules for values and memory; the intuition we present here in our proof sketch is that security is not relevant in the memory; as you can see from the grammar in Fig. 6, we keep the security level on the value (outside of the memory), so the result of a high method call with  $\rightarrow_{\text{call}}$  is well typed because it is only concerned with the non-security aspects of the type system, and the security level is tracked during reduction. See how, for example, in rule Call of Fig. 4, the security level and modifier of the parameters can be promoted before inlining the method body.  $\square$

To prove the noninterference Theorem 5.10, we use the property that the reduction terminates (Lemma 5.8) and the bisimulation core that each reduction step meets the requirements of noninterference (Lemma 5.9). We prove, if two memories that are similar and both expressions reduce, the new memories still have to be similar and the reduced values are equal w.r.t. **low** values. This Theorem 5.10 has the same shape as Theorem 5.0, but uses the definitions for similarity of memories (Def. 5.5) and expressions (Def. 5.6). As we said before, we exclude declassification in the proof. Therefore, we cannot guarantee security for programs with **declassify** expressions. Related works generalize the noninterference property to provide stronger guarantees for programs including declassification [Sabelfeld and Sands 2009].

#### THEOREM 5.10 (NONINTERFERENCE).

*If we have expressions  $e_1$  and  $e_2$  without declassification that are well typed and  $e_1 \text{ lowEqual } e_2$ ,  $M_1$  and  $M_2$  are well typed memories,  $M_1 \text{ similar}(e_1) M_2$ ,  $M_1|e_1 \rightarrow^* M'_1|v_1$ , and  $M_2|e_2 \rightarrow^* M'_2|v_2$  then  $M'_1 \text{ similar}(v_1) M'_2$ ,  $v_1 \text{ lowEqual } v_2$ , and  $M'_1, M'_2, v_1$ , and  $v_2$  are well typed.*

PROOF. From Lemma 5.8 and  $M_1|e_1 \rightarrow^* M'_1|v_1$  and  $M_2|e_2 \rightarrow^* M'_2|v_2$ , we know that  $M_1|e_1 \rightarrow_{call} M'_1|v_1$  and  $M_2|e_2 \rightarrow_{call} M'_2|v_2$ . Then, by induction on the number of steps of  $\rightarrow_{call}$ :

Base:  $e_1 \text{ lowEqual } e_2 \text{ lowEqual } v_1 \text{ lowEqual } v_2$ ,  $M_1 = M'_1$ ,  $M_2 = M'_2$ . Thus,  $M'_1 \text{ similar}(v_1) M'_2$  holds because  $M_1 \text{ similar}(e_1) M_2$ .

Inductive step: By Lemma 5.9 (bisimulation core) and the inductive hypothesis, each reduction step establishes similar memories  $M'_1$  and  $M'_2$ , computes *lowEqual* expressions, and preserves that only necessary values are in the **high** ROG.  $\square$

## 6 TOOL SUPPORT AND EVALUATION

In this section we present tool support for SIFO and evaluate feasibility of SIFO by implementing five case studies. Additionally, we benchmark precision and recall of the information flow analysis by adapting the IFSpec benchmark suite [Hamann et al. 2018] to SIFO.

### 6.1 Tool Support

We implement SIFO as a pluggable type system for L42 [Giannini et al. 2019]. L42 is a pure object-oriented language with a rich type system supporting the type modifiers used by SIFO.

Conveniently, L42 allows pluggable type systems [Andreae et al. 2006; Papi et al. 2008] to add an additional layer of typing. We add rules to support the typing of expressions with security levels. Both Java and L42 supports pluggable type systems using annotations: type names preceded by the symbol @. In our SIFO library, these annotations are used to introduce the security levels.<sup>5</sup>

Some changes of SIFO are necessary to comply with L42: L42 supports the uniform access principle [Meyer 1988]; thus there is no dedicated syntax for field assign and field access, but they are modeled by getters and setters. Additionally, the constructor does not have dedicated syntax, but it is a static method with return type `This`. In this way, we only need to type check method calls. Moreover, this allows more flexibility since multiple method types are now transparently and consistently applied in all those cases. We also had to extend our type system to support the features of L42. While adding loops and other conventional constructs was trivial, we had to be careful while extending our type system to support exceptions, since exceptions constitute yet another way for a method execution to propagate secret information to an observer. Thus, we consider exceptions similar to a return type.

Additionally, an exception prevents the execution of the code after it was thrown. Thus, after the exception is caught, the program may collect information about when the execution was interrupted in order to discover what expression raised the exception. This is another option to propagate secret information to an observer. Our current extension supporting exceptions is quite conservative, requiring the use of a single security level for all free variables, exceptions, and results of a try-catch block. In future work, we plan to formalize the extension with exceptions more precisely.

### 6.2 Feasibility Evaluation

To evaluate the feasibility of SIFO, we implemented four case studies from the literature in SIFO: Battleship [Stoughton et al. 2014; Zheng et al. 2003], Email [Hall 2005], Banking [Thüm et al. 2012], and Paycard (<http://spl2go.cs.ovgu.de/projects/57>). Additionally, we implemented a novel case study of our own, the *Database*. The metrics of the case studies are shown in Table 1.

**6.2.1 Battleship.** Our evaluation is focused on the *Battleship* case study because this program is carefully described by Stoughton et al. [Stoughton et al. 2014] as a general benchmark to evaluate

<sup>5</sup>You can find a version of L42 with our SIFO library and the case studies at <https://l42.is/SifoArtifactLinux.zip> and <https://l42.is/SifoArtifactWin.zip>. This also contains more information about the detailed syntax in the readme.



Name	#Security Levels	#Security Annotations	#Lines of Code
Battleship	4 (+2 generic)	21 (208 in Jif)	431 (611 in Jif)
Database	4	6	73
Email	2	20	260
Banking	2	20	127
Paycard	2	15	95

Table 1. Metrics of the case studies

information flow control. Moreover, this case study is also implemented in Jif<sup>6</sup>, thus allowing us to directly compare their results with our work.

*Battleship* is the implementation of a two player board game. At the start, each player places a fixed number of ships of varying length on their private board. The board has a two-dimensional grid. The players only know the placement on their board and have to guess where the other player placed the ships. During the game, the players take turns and *shoot* cells on the board of the opponent to sink ships. The first player wins the game who sinks all opponent's ships.

Thanks to our flexible SIFO type system, we implemented most of the code without any security annotations. We wrote a generic `PlayerTrait` trait that is parameterized over the security levels *SelfL* and *OtherL* to distinguish both players. Our implementation of *Battleship* uses many features from full L42, not just the minimal core presented in this paper. In particular, we rely on L42 traits and their encoding for generics. We will expand on this in Section 7.

Our `ExampleGame` class implements a mutually distrustful player scenario [Stoughton et al. 2014]. In this setting, even if one of the two players is replaced with an adversarial player, we ensure that only a correctly executed game will terminate without exception. This scenario highlights nicely the properties of our system: we ensure noninterference that an adversarial player is unable to read the opponent's board state. In `ExampleGame`, the two players *Player1* and *Player2* create the boards and shoot consecutively. We have to ensure that each player creates a confidential board that the other player cannot read. In the concrete implementation, we annotate the board with the security level of one of the players to restrict readability. Deep immutability enforces that the board is not manipulated during the game. Then, each player gets a reference to the confidential board of the opponent. Contrast this with the Jif implementation of the same game: Jif uses the concept of label expressions that specify the allowed readers and writers of objects. In Jif, boards can be read by only one player, but they are trusted by both players. The first player creates a board, the second player endorses this board, and the first player then saves this board that is trusted by both players. Endorsement is the name of downgrading integrity, similar to declassification for confidential data. The endorsement of the board is implemented in Jif with defensive cloning. A new trusted board is created, and all ships on the input board are cloned to add them to the trusted board. In SIFO, the endorsement and defensive cloning is not necessary because we can rely on deep immutability of the type system that prevents manipulation of the boards.

When a player shoots, it has to be correctly revealed if it was a miss, a hit, or a hit that sunk a specific ship. The process of one shooting round is shown in Listing 4. The method `round` has as parameter which is the opposing player. In Line 2, there is a dynamic check for validity of the game.

<sup>6</sup>See [Zheng et al. 2003] found on the Jif website <https://www.cs.cornell.edu/jif/>

```

1  mut method Bool round(mut OtherPlayer other) = (
2    X[this.myRound(); !this.win(this.myShots()); !this.win(this.otherShots())]
3    coord = this.fire()
4    @OtherL FireResult res = this.board().fire(coord=coord, shots=this.myShots())
5    ResultSigner s = this.signer()
6    @OtherL ResultSigner.Signed signed = s(label=coord.toS(), data=res)
7    ResultSigner.Signed freeSigned=other.declassify(signed)
8    X[this.signer().mine(freeSigned, label=coord.toS())]
9    r = freeSigned.data().answer().repr()
10   this.myShots(\myShots.with(row=coord.row(), col=coord.col(), val=r))
11   this.myRound(Bool.false())
12   this.win(this.myShots())
13 )

```

Listing 4. Implementation of one shooting in SIFO

`x` works like `assert` in Java. It must be the round of the player, and the game must not be yet won by any player. The method `fire` in Line 3 asks for the next coordinates to shoot. These coordinates are used to check for the result of the shot on the board. As this is the board of the opponent, the result is labeled with the opponent's security level (Line 4). We now have to declassify the result to be able to read it, but only the opponent has the right to declassify the result of such shot. An adversarial opponent could manipulate the declassified result, we therefore use a signing mechanism to exclude manipulation. Thus, the confidential result is signed by the shooting player and send to the opponent (Line 6), so that the opponent declassifies the result (miss, hit, ship sunk) in Line 7. In Line 8, it is checked that the correct signed result is returned. In our implementation, we can rule out manipulation as the correct result is immutable and a newly created result by the opponent cannot have the signature of the shooting player. The shot and the result are then added to a list for validating subsequent rounds (e.g., in Line 12 to check whether the game is won). Then, the opponent player takes turn. If a game rule violation is detected during the game (e.g., a manipulated result of a shot), the game can be aborted by either player.

In Jif, a player must trust that the result of a shot is correctly revealed by the opponent. In Jif, it would be easy to implement a `BadPlayer` that returns an incorrect result of a shot, as there is no check for a manipulation [Stoughton et al. 2014]. Additionally, the Jif implementation uses defensive cloning when passing the coordinates of a shot to the opposing player.

We now compare both implementation on a more general level. Most classes are written parameterized with a security level  $L$  in Jif. In SIFO, we write classes like `Ship` without any security annotation, but we are able to use them in secure contexts with our promotion rules. With this technique, we are able to write only 21 security annotations in the whole implementation. In Jif, we count 208 annotations.

For the creation of players, we have a similar concept as Jif. While Jif used a generic `Player` class, we created a generic `PlayerTrait` trait with two security levels `SelfL` and `OtherL`. This generic trait is instantiated as `PlayerTrait['SelfL=>Player1; 'OtherL=>Player2]` which can then be used to create object instances. In Jif, `[Player1, Player2] player1 = new Player[Player1, Player2]();` is written to create a new player.

In summary, we implemented *Battleship* in SIFO by relying on our promotion rules and the immutability of trustworthy objects. Thanks to preexisting L42 features, we have not needed the complex expressiveness of Jifs label expression, which is also discussed in Section 7.

6.2.2 *Database.* The *Database* is a system where two databases are not allowed to interfere. Through the different security levels, we ensure that a value read from one database cannot be inserted into the other one. This can be obtained just by annotating the `Gui` class with six security annotations, as shown in Listing 5.

```

1 Gui={
2   mut @Left Database dbLeft
3   mut @Right Database dbRight
4   class method mut This (mut @Left Database dbLeft, mut @Right Database dbRight)
5   class method mut This ()=(
6     capsule @Left Database dbl=Database(name="left",rows=Rows())
7     capsule @Right Database dbr=Database(name="right",rows=Rows())
8     This(dbLeft=dbl, dbRight=dbr)
9   )
10  }
```

Listing 5. Gui implementation in SIFO

The other classes are implemented without any security level. This is possible because the class `Gui` is the only class that uses databases with different security levels. This means that the actual database code is all free from security annotations; as you can see above different instances of `low capsule Database` can be transparently promoted to different security levels `Left` and `Right`.

6.2.3 *Further case studies.* The *Email* system ensures that encrypted emails are only decrypted if the public and private key pair used is valid. It also guarantees that private keys are not leaked. The `Email` system needed only 20 security annotations in 260 lines of code.

*Banking* and *Paycard* are two systems that represent payment systems where it is crucial that the calculations of new balances are correct and information is not leaked. By setting the balance to `high` and checking that the system is typable, we are certain that the balance is not leaked to attackers. For *BankAccount* and *Paycard* 20 and 15 security annotations were needed with 127 and 95 lines of code.

6.2.4 *Discussion.* Code following a pure object-oriented style is often directly supported by SIFO without any special adaptation. However, when updates to local variables and statements/conditionals are used, the programmer may have to rely on some simple programming patterns: for example, in a conditional, we cannot directly update a `high` field of a `low` object, as `low` objects cannot be manipulated in `high` conditional statements. This limitation can be circumvented by wrapping the updatable field into a mutable proxy object (e.g., `o.proxy.field` instead of `o.field`). Such a proxy object can then be saved in a `high` temporary variable; and such a variable can be used to manipulate the state. We have to use the `high` proxy object because our IR-rule is slightly conservative. It does not check if only `high` fields of a `low` object are manipulated. This is still easier in comparison to Jif because in Jif, the object instance has to be cloned so that the user keeps a reference to the `low` object and can manipulate the `high` field data with the cloned instance.

Another insight is that major parts of the case studies could be written without any use of security levels because the multiple method types promote the parameters of a called method to the required security levels in necessary cases. This allowed us to write secure programs relying on libraries and data structures without any security annotation. In our case study, we could reuse a list implementation and securely promote it to any security level if needed. The type system then checks that instantiated lists of different security levels did not interfere. For example, a `high` list can only contain objects of at least a `high` security level. Moreover, we were also able to encode domain-specific data structures and functionality without any security annotation. In Jif,

this requires generic classes written with security annotations in all cases. Information flow can be enforced by annotating just the few method bodies that put separate systems into communication.

Major parts of the case studies were implemented without the use of `declassify`. We only needed it in *Battleship* to declassify shot results as intended by the game. Additionally, we declassified console output at the end of program execution in the other case studies to print results for the user.

By explicitly typing references as `imm` or `capsule`, the code quality increases, because programmers can rely on properties which are enforced by the type system. Furthermore, the security levels serve as active documentation for the programmer. In the *Database* example, we know which database a value comes from by reading the security level. To reduce the writing effort for programmers, sensible defaults are useful: if a security level is not specified, `low` is used.

### 6.3 Benchmarking with IFSpec

To evaluate precision and recall of SIFO, we applied SIFO to the IFSpec benchmark suite [Hamann et al. 2018]. IFSpec contains 80 samples to test information flow analysis tools. In addition to the core samples, 152 samples from the benchmark suite SecuriBench Micro<sup>7</sup> are adapted and integrated into IFSpec. The samples are all available in Java and Dalvik. To benchmark SIFO, we translated the 80 core sample when it was possible. Samples that used Java specific features were not translated. In total 40 samples are implemented in SIFO. For these samples, we compare SIFO with Cassandra [Lortz et al. 2014], JOANA [Graf et al. 2013], JoDroid [Mohr et al. 2015], KeY [Ahrendt et al. 2016], and Co-Inflow [Xiang and Chong 2021] (with and without additional security annotations) which were all evaluated before with IFSpec.

Each sample is labeled as secure or insecure. When a sample contains a leak and a tool reports a leak, we categorize it as true positive (TP). When a sample contains no leak and a tool reports no leak, we categorize it as true negative (TN). When a sample contains no leak but a tool reports a leak, we categorize it as false positive (FP). When a sample contains a leak but a tool reports no leak, we categorize it as false negative (FN). From these four categories, we can calculate precision and recall of the tools. The recall is computed as:  $\#TP / (\#TP + \#FN)$ . Recall determines the percentage of samples correctly classified as insecure considering all samples containing a leak. The precision is computed as:  $\#TP / (\#TP + \#FP)$ . Precision determines the percentage of samples correctly classified as insecure considering all samples classified as insecure by the tool.

In Table 2, we show the results of the benchmarking. All six tools found the 18 samples containing a leak. This results in a recall of 100% for all tools. No tool classified a sample false negative. Regarding precision, the tools have slight differences. Cassandra and Co-Inflow without additional annotations have the lowest precision of 54.5%. JOANA has the highest precision of 62.1%, but Co-Inflow has a higher precision of 81.8% if additional security annotation is given by the programmer. SIFO has a precision of 58.1%.

*Discussion of the False Positive Samples.* With SIFO, 13 samples are typed as insecure, which are labeled as secure by the authors of the benchmark. We will classify these samples into categories to discuss the result of SIFO. For six samples, the type system of SIFO is not precise enough to recognize that the sample is secure. For example, if in a conditional expression both branches assign the same value to a `low` reference, we pessimistically dismiss this program.

Three samples are constructed to introduce a leak which is overwritten in the end. A simple example is that a secret value is assigned to a public variable and in the next line, the public variable is overwritten. We clearly prohibit the first assignment of the secret value. These examples are constructed for taint analysis tools and are not suitable for type systems.

<sup>7</sup><https://github.com/too4words/securibench-micro>

<b>Tool</b>	<b>#Samples</b>	<b>TP</b>	<b>TN</b>	<b>FP</b>	<b>FN</b>	<b>Recall</b>	<b>Precision</b>
Cassandra	40	18	7	15	0	100%	54.5%
JOANA	40	18	11	11	0	100%	62.1%
JoDroid	40	18	9	13	0	100%	58.1%
KeY	40	18	8	14	0	100%	56.3%
Co-Inflow	40	18	7	15	0	100%	54.5%
Co-Inflow- Annotations	40	18	18	4	0	100%	81.8%
SIFO	40	18	9	13	0	100%	58.1%

Table 2. Overview of the benchmark results

One sample is labeled as secure because in the provided code there is no way to access the secret values. In sample `Webstore`, a secret and a public value are assigned to a public list and only the public value is accessed in the code. When a simple getter-method for the secret value is added, the sample would be insecure. As our type system is modular, we directly prohibit the assignment of secret values to the public list. We do not check that there is currently no available method to access the stored value.

For three samples, again the modular reasoning of the type system pessimistically rejects secure programs. In `DeepCall`, a chain of method calls is insecure because the first secret value is propagated through all calls and returned as a public value. In the similar sample `DeepCall2`, the last call always returns the same value independent of the secret input value. This sample is considered secure. Our types system does not check all method calls globally, it just reasons that a secret value is passed to the next method and that a secret return value is expected. That the secret return value is actually a public constant in the sample `DeepCall2` is outside of the modular reasoning.

Most examples that SIFO rejects are constructed by developers to contain a security problem which is erased or not accessible in the remaining code of the sample. As our type system prohibits any introduction of security violations, we reject these samples. To support this statement, we rewrote eight of the 13 false positive samples to be semantically similar and accepted by SIFO.

## 7 RELATED WORK

Static and dynamic program analysis [Austin and Flanagan 2009; Nielson et al. 1999; Russo and Sabelfeld 2010; Zhang et al. 2015], as well as security type systems [Banerjee and Naumann 2002; Ferraiuolo et al. 2017; Hunt and Sands 2006; Li and Zhang 2017; Simonet 2003; Volpano et al. 1996] are used to enforce information flow policies. We refer to Sabelfeld and Myers [2003] for a detailed overview.

*Taint Analysis.* Taint analysis [Arzt et al. 2014; Enck et al. 2014; Hedin et al. 2014; Huang et al. 2014, 2012; Milanova and Huang 2013; Roy et al. 2009] is a related analysis technique that detects direct information flows from tainted sources to secure sinks by analyzing the assignments of variables and fields. Those taint analysis works do not provide a soundness property, while the SIFO noninterference proof guarantees the security of type checked code. Except for JSFlow [Hedin et al. 2014], Cassandra [Lortz et al. 2014], JOANA [Graf et al. 2013], and JoDroid [Mohr et al. 2015], these related works do not cover implicit information flows through conditionals, loop statements,

or dynamic dispatch. SIFO also detects implicit information flows through dynamic dispatch (conditional and loop statements are not in the core language, but included in our implementation). Crucially, the noninterference proof of SIFO relies on detecting implicit information flow.

Coarse-grained dynamic information flow approaches [Jia et al. 2013; Nadkarni et al. 2016; Xiang and Chong 2021] track information at the granularity of lexically or dynamically scoped section of code. Instead of labeling every value individually, coarse-grained approaches label an entire section with one label. All produced values within this scope implicitly have that same label. Therefore, the writing effort for developers to annotate programs is reduced. To still obtain good results of the information flow analysis, for example, Xiang and Chong [Xiang and Chong 2021] introduce opaque labeled values to permit labeled values where programmers have not provided a label. If no further annotation is given by the programmer, the precision of the information flow analysis can be decreased. As the evaluation shows, Co-Inflow [Xiang and Chong 2021] has better precision when the programmer annotates the program. However, the precision is not a limitation of coarse-grained approaches compared to fine-grained approaches. Type systems for fine- and coarse-grained information flow control are equivalent in terms of precision as shown by Rajani et al. [Rajani et al. 2017; Rajani and Garg 2018]. For dynamic information flow control mechanisms, Vassena et al. [Vassena et al. 2019] have similar results.

The work by Huang, Milanova et al. [Huang et al. 2014, 2012; Milanova and Huang 2013] is closely related to our approach because viewpoint adaption with polymorphic types is similar to our  $mdf \triangleright mdf'$  operator for type modifiers. For field accesses, the type of the accessed object depends on the reference and the field type. They use read-only references to improve the precision of their static analysis technique by allowing subtyping if the reference is read-only. In SIFO, we also use deep immutable and capsule references, extending the expressiveness of our language.

*Comparison to Jif.* In this work, we explored the specific area of secure type systems for object-oriented languages [Banerjee and Naumann 2002; Barthe et al. 2007; Barthe and Serpette 1999; Myers 1999; Sabelfeld and Myers 2003; Strecker 2003; Sun et al. 2004]. The most important work to compare against is Jif [Myers 1999] (see Section 2). In this paper, we presented a minimal core of SIFO for the soundness and noninterference proofs. Nonetheless, we compare SIFO with Jif by discussing their common and different features. A main difference is the handling of aliases: Jif does not use any kinds of regions or alias analysis to reason about bounded side effects. Therefore, Jif pessimistically discards many programs introducing aliases (see the example in Section 2.1 that is not typable in Jif). On the other hand, SIFO restricts the introduction of insecure aliases and is therefore able to safely type more programs. SIFO's expressiveness relies on the safe promotion of `imm` or `capsule` references. As shown in Section 2 and in Section 6.2, programs can be typed securely without defensive cloning [Bloch 2016] because `imm` and `capsule` modifiers allow promoting objects to higher security levels. In Jif, a similar promotion is only allowed for primitive types.

The SIFO type system leverages on a minimalistic syntax of security annotation, where types contain a security level. Jif offers a much more elaborated syntax: in Jif, a security label is an expression consisting of a set of policies [Myers and Liskov 2000]. Each policy has an owner  $o$  and a set of readers  $r$ . For example, the label  $o_1 : r_1; o_2 : r_1, r_2$  states that the policy of  $o_1$  allows  $r_1$  to read the value and  $o_2$  allows  $r_1$  and  $r_2$  to read the value. Hence,  $r_1$  is the only reader to fulfill both policies. These label expressions get more complicated, the more policies are conjoined, but a programmer gets more flexibility to express fine-grained access restrictions.

SIFO has a similar expressiveness to Jif, but does not need to resort to such complex label expressions. To show this, consider the following scenario from Jif [Myers and Liskov 2000]: A person Bob that wants to create his tax form using an online service. In the scenario, Bob wants to prevent his information from being leaked to the online service, and the provider of the



```

1  class Protected {
2      final label{this} lb;
3      Object{*lb} content;
4      public Protected{LL}(Object{*LL} x, label LL) {
5          lb = LL;
6          super();
7          content = x;}
8      public Object{*L} get(label L){L} throws (IllegalAccessException) {
9          switch label(content) {
10             case (Object{*L} unwrapped) return unwrapped;
11             else throw new IllegalAccessException();}}
12     public label get_label() {
13         return lb;}}

```

Listing 6. Class Protected in Jif with security parameterization [Myers 1999]

service does not want its technology and data to be leaked in the process of generating the tax form. This constraint is related to the mutually distrustful players of the *Battleship* case study. To comply with these constraints, Bob labels his data with *bob : bob* and sends it to the online service provider. The provider, labels its own data with *provider : provider*, so the calculated tax has the label *bob : bob; provider : provider*. This result cannot be read because the labels disagree on their reader sets. To release the information to Bob, the provider declassifies the label by removing the provider policy. As only the final tax form is declassified, the released information from the provider is limited. The final tax form with the label *bob : bob* is then sent to Bob.

In SIFO, we can handle the same scenario as follows: Bob wants to protect his private information, so he can set the security level to **bob**, but he can also set a type modifier. With **read**, he ensures that his data cannot be manipulated and integrated into the provider's data. With **imm**, only the manipulation is prevented. If Bob trust the provider, he can send a **capsule** object to the provider. The provider can then manipulate and alias the data, but Bob is sure, that the manipulation is restricted to only the data reachable from the given reference. In the case of the provider, they get a reference to the data of Bob with a security level and a type modifier. In the most restricted case of a **read** reference, the provider can still use the information and calculate the final tax form, but a manipulation or aliasing of Bob's data is prevented. The security level of the result is the least upper bound of **bob** and **provider**. To declassify the results for Bob, the final tax form needs to be **imm** or **capsule** to allow safe sharing or transfer of the confidential data.

With this example, we discuss the secure transfer of data. In SIFO, by using a **read**, **imm**, or **capsule** modifiers, Bob specifies how the information is usable. In Jif, the label **bob: bob** does not restrict the use in the same way. There is no language support to ensure that a unique portion of store is transferred to the provider. There is also no guarantee that the data is not manipulated, as with **imm**. In Jif, if the provider has a read permission for Bob's data, they can freely manipulate it. Furthermore, if the provider wants to ensure that they are the current owner of Bob's tax data, they have to clone the data (defensive cloning [Bloch 2016]).

To grasp the difference in the annotation burden, consider the Jif example in Listing 6 of a class that protects data from insecure access. Jif uses a parameterized label system where a class or methods have a generic label *L*. The label *L* can be initialized with any specific security level. This places a large conceptual burden on the programmer, as the label *L* is used in every field and method of the class. Additionally, no legacy code can be used that is not parameterized properly.



SIFO encourages a style where most code is completely clear of any security annotation; in particular, most algorithms and most common data types like collections does not need any kind of security annotations at all. Only code that is explicitly and directly involved in the handling of security-critical data needs to be written with security in mind. Unlabeled classes and methods are implicitly annotated with the lowest security level. Thanks to the flexibility of multiple method types, they can be safely promoted to any higher level.

Jif has additional features that are not presented in the core of SIFO. The SIFO core works with a finite lattice of security levels instead of the complex label expressions in Jif [Myers and Liskov 2000] with an infinite set of possible labels. Thanks to the embedding in L42, we get label polymorphism for free by relying on L42 encodings for generics. Thus, on one side SIFO allows to remove the complexity of having most of the labels generic, on the other side when generic labels are truly needed (for example to write code that have to work on unknown labels) we can rely on the L42 generics encoding, as we do in the BattleShip example.

Jif has dynamic checks of security labels. See Line 9 in Listing 6 where the security level of the object content is checked. This feature can be emulated in SIFO with the following programming pattern. Any is the equivalent of Object in Java.

```

1 BoxLeft=Data:{@Left Any left}
2 BoxRight=Data:{@Right Any right}
3 ...
4 low Any a
5 if a <:BoxRight return a.right()
```

In a more concrete example, a Person Bob creates a BoxLeft or BoxRight object with the secure data in the field. This object is then sent as `low Any a` to Alice and Alice can discover with `instanceof (<: in L42)` if it is a BoxLeft or a BoxRight object. As you can see, by knowing the explicit type, we know also the security level of the data in the field. Thus, by adding an explicit boxing step, we enable the users to handle any kind of label and to dynamically check on those.

Jif has robust declassification [Chong and Myers 2006] which means that an attacker is not able to declassify information, or to influence what information is declassified by the system that is above the security level that the attacker is allowed to read. In the full embedding in L42, declassification can be sealed behind the object capability model, as we did in the *Battleship* case study. The L42 object capability model is flexible and can provide a range of useful guarantees [Miller 2006]. Indeed, you can see the *Battleship* case study as an exemplar representation of robust declassification. Even if we replace one of the player with adversarial code, such code will not be able to declassify the opposing board; even while holding a reference to such a board.

In future work, we want to extend SIFO to work with any partial order of security levels as discussed in Section 9. With this feature, we are closer to the expressive power of Jifs label expressions.

*Other Information Flow Techniques.* Hoare-style program logics are also used to reason about information flow. The work of Andrews and Reitman [Andrews and Reitman 1980] encodes information flow in a logical form for parallel programs. Amtoft et al. [Amtoft et al. 2006; Amtoft and Banerjee 2004] use Hoare-style program logic and abstract interpretation to analyze information flow. This approach is the basis in SPARK Ada for specifying and checking information flow [Amtoft et al. 2008]. For Java, Beckert et al. [Beckert et al. 2013] formalized the information flow property in a programming logic using self-composition of programs and an existing program verification tool to check information flow. Similarly, Barthe et al. [Barthe et al. 2004] and Darvas et al. [Darvas et al. 2005] analyze the information flow by using self-composition of programs and standard software verification systems. Terauchi and Aiken [Terauchi and Aiken 2005] combined a self-composition

technique with a type system to profit from both techniques. Küsters et al. [Küsters et al. 2015] propose a hybrid approach by using JOANA [Graf et al. 2013] and verification with KeY [Ahrendt et al. 2016] to check the information flow.

The related IFbC approach by Schaefer et al. [Runge et al. 2020; Schaefer et al. 2018] ensures information flow-by-construction. Here, the information flow policy is ensured by applying a sound set of refinement rules to a starting specification. Instead of checking the security after program creation, the programmer is guided by the rules to never violate the policy. Compared to SIFO, their approach is limited to a while language without objects.

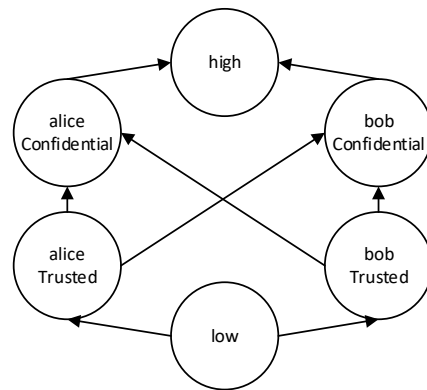
## 8 CONCLUSION

In this work, we presented a type system of an object-oriented language for secure lattice-based information flow control using type modifiers that detects direct and implicit information flows. This language supports secure software development by enforcing noninterference. We leverage previous work on immutability and encapsulation to greatly increase the expressive power of our language. Additionally, promotion/multiple method types encourage reusability of secure programs without burdening the developer. We formalized the secure type system, proved noninterference, and showed feasibility by implementing SIFO as a pluggable type system for L42, and conducting an evaluation with several case studies. In the future, we want to formalize exceptions in SIFO to extend the expressiveness of the language. We also want to generalize the proof to include declassification. Furthermore, we could reduce the typing effort of programmers by introducing type inference.

## 9 FUTURE WORK: INTEGRITY AND CONFIDENTIALITY

As noted by Biba [Biba 1977] integrity can be seen as a dual to confidentiality, which means that either of them can be checked with the same information flow analysis techniques. For confidentiality, information must not flow to inappropriate destinations; dually, for integrity, information must not flow from inappropriate sources. In this work, we made all our discussion about confidentiality. If a user of SIFO is instead interested in integrity, they can simply use our type system with any lattice of integrity levels. However, it is also possible to track both properties at the same time: The trick is to not rely too much on data sources with the lowest or highest security level (e.g. **low** and **high**): since **high** can see all the information, **high** offers no integrity. In the same way, **low** can write to all the information, thus **low** data needs to be always intrinsically valid/trusted. Note that we can still declare **low** fields and **low** data structures, it is sufficient for sensitive data to be stored somewhere nested inside the ROG from a non **low** reference, as we shown with the database case study.

In this work, we assumed a lattice of security levels. However, Logrippo [Logrippo 2018] has proposed that just a partially ordered set would be appropriate to model security. If we allowed just a partial order of security levels, SIFO would allow to encode both integrity and confidentiality at the same time instead of using two lattices for confidentiality levels and integrity levels. Consider the following example, where Bob and Alice have both confidential and trusted data. We can define a partially ordered set as shown on the right.



Integrity: **aliceTrusted/bobTrusted** is data that Alice/Bob trust to be valid. For example, only **low** and **bobTrusted** can write on **bobTrusted** data. Confidentiality: **aliceConfidential/bobConfidential** is data

that Alice/Bob wants to keep private. For example, `low`, `aliceTrusted`, `bobTrusted`, and `bobConfidential` can write on `bobConfidential`, but `bobConfidential` can only be read by `bobConfidential` and `high`. Those security levels also imply that `bobTrusted` can be read by `bobTrusted`, `aliceConfidential`, `bobConfidential`, and `high`.

Being able to express integrity and confidentiality at the same time with the same lattice is clearly a great advantage; however we are still investigating if supporting partially ordered sets instead of a lattice would have subtle consequences that interfere with our noninterference property.

*Acknowledgments.* This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs (46.23.03).

## REFERENCES

- Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Lecture Notes in Computer Science, Vol. 10001. Springer.
- Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. 2006. A Logic for Information Flow in Object-Oriented Programs. In *POPL*. 91–102.
- Torben Amtoft and Anindya Banerjee. 2004. Information Flow Analysis in Logical Form. In *SAS (LNCS, Vol. 3148)*. Springer, 100–115.
- Torben Amtoft, John Hatcliff, Edwin Rodríguez, Robby, Jonathan Hoag, and David A. Greve. 2008. Specification and Checking of Software Contracts for Conditional Information Flow. In *FM*. Springer, 229–245.
- Chris Andrae, James Noble, Shane Markstrum, and Todd Millstein. 2006. A Framework for Implementing Pluggable Type Systems. In *OOPSLA*. 57–74.
- Gregory R. Andrews and Richard P. Reitman. 1980. An Axiomatic Approach to Information Flow in Programs. *TOPLAS* 2, 1 (1980), 56–76.
- Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Oceau, and Patrick D. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *PLDI*, Vol. 49. ACM, 259–269.
- Thomas H Austin and Cormac Flanagan. 2009. Efficient Purely-Dynamic Information Flow Analysis. In *PLAS*. ACM, 113–124.
- Anindya Banerjee and David A Naumann. 2002. Secure Information Flow and Pointer Confinement in a Java-like Language.. In *CSFW*, Vol. 2. 253.
- Gilles Barthe, Pedro R D’Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *CSF*. IEEE, 100–114.
- Gilles Barthe, David Pichardie, and Tamara Rezk. 2007. A Certified Lightweight Non-Interference Java Bytecode Verifier. In *European Symposium on Programming*. Springer, 125–140.
- Gilles Barthe and Bernard P Serpette. 1999. Partial Evaluation and Non-Interference for Object Calculi. In *FLOPS*, Vol. LNCS. Springer, 53–67.
- Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H Schmitt, and Mattias Ulbrich. 2013. Information Flow in Object-Oriented Software. In *LOPSTR*, Vol. LNCS. Springer, 19–37.
- D Elliott Bell and Leonard J La Padula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation*. Technical Report. MITRE Corp Bedford MA.
- Kenneth J Biba. 1977. *Integrity Considerations for Secure Computer Systems*. Technical Report. MITRE Corp Bedford MA.
- Joshua Bloch. 2016. *Effective Java*. Pearson Education India.
- Stephen Chong and Andrew C Myers. 2006. Decentralized Robustness. In *19th IEEE Computer Security Foundations Workshop (CSFW’06)*. IEEE, 12–pp.
- Ádám Darvas, Reiner Hähnle, and David Sands. 2005. A Theorem Proving Approach to Analysis of Secure Information Flow. In *SPC*, Vol. LNCS. Springer, 193–209.
- Dorothy E Denning. 1976. A Lattice Model of Secure Information Flow. *CACM* 19, 5 (1976), 236–243.
- William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *TOCS* 32, 2, Article 5 (June 2014), 29 pages.
- Andrew Ferraiuolo, Weizhe Hua, Andrew C Myers, and G Edward Suh. 2017. Secure Information Flow Verification with Mutable Dependent Types. In *DAC*. IEEE, 1–6.

- Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. 2019. Flexible Recovery of Uniqueness and Immutability. *Theoretical Computer Science* 764 (2019), 145–172.
- Joseph A Goguen and José Meseguer. 1982. Security Policies and Security Models. In *S&P*. IEEE, 11–11.
- Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc.
- Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. 47, 10 (2012), 21–40. <https://doi.org/10.1145/2398857.2384619>
- Jürgen Graf, Martin Hecker, and Martin Mohr. 2013. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13) (Lecture Notes in Informatics (LNI) 215)*. Springer, 123–138.
- Robert J Hall. 2005. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering* 12, 1 (2005), 41–79.
- Tobias Hamann, Mihai Herda, Heiko Mantel, Martin Mohr, David Schneider, and Markus Tasch. 2018. A Uniform Information-Flow Security Benchmark Suite for Source Code and Bytecode. In *Nordic Conference on Secure IT Systems*. Springer, 437–453.
- Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking Information Flow in JavaScript and Its APIs. In *SAC (Gyeongju, Republic of Korea)*. ACM, 1663–1671.
- Wei Huang, Yao Dong, and Ana Milanova. 2014. Type-based Taint Analysis for Java Web Applications. In *FASE (LNCS, Vol. 8411)*. Springer, 140–154.
- Wei Huang, Ana Milanova, Werner Dietl, and Michael D Ernst. 2012. ReIm & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. *ACM SIGPLAN Notices* 47, 10 (2012), 879–896.
- Sebastian Hunt and David Sands. 2006. On Flow-Sensitive Security Types. *SIGPLAN Not.* 41, 1 (Jan. 2006), 79–90.
- Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *TOPLAS* 23, 3 (2001), 396–450.
- Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-time Enforcement of Information-Flow Properties on Android. In *European Symposium on Research in Computer Security*. Springer, 775–792.
- Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. 2015. A Hybrid Approach for Proving Noninterference of Java Programs. In *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE, 305–319.
- Peixuan Li and Danfeng Zhang. 2017. Towards a Flow-and Path-Sensitive Information Flow Analysis. In *CSF*. IEEE, 53–67.
- Luigi Logrippo. 2018. Multi-level Access Control, Directed Graphs and Partial Orders in Flow Control for Data Secrecy and Privacy. In *Foundations and Practice of Security*. Springer International Publishing, 111–123.
- Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. 2014. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. 93–104.
- Bertrand Meyer. 1988. Eiffel: A Language and Environment for Software Engineering. *Journal of Systems and Software* 8, 3 (1988), 199–246.
- Ana Milanova and Wei Huang. 2013. Composing Polymorphic Information Flow Systems with Reference Immutability. In *FTfJP*. ACM, Article 5, 7 pages.
- Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.
- Martin Mohr, Jürgen Graf, and Martin Hecker. 2015. JoDroid: Adding Android Support to a Static Information Flow Control Tool. In *Software Engineering (Workshops)*. Citeseer, 140–145.
- Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *POPL (San Antonio, Texas, USA)*. ACM, New York, NY, USA, 228–241.
- Andrew C Myers and Barbara Liskov. 2000. Protecting Privacy Using the Decentralized Label Model. *TOSEM* 9, 4 (2000), 410–442.
- Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC Enforcement on Android. In *25th USENIX Security Symposium (USENIX Security 16)*. 1119–1136.
- Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer.
- Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. 2008. Practical Pluggable Types for Java. In *ISSTA*. 201–212.
- Benjamin C Pierce. 2002. *Types and Programming Languages*. MIT press.
- Vineet Rajani, Iulia Bastys, Willard Rafnsson, and Deepak Garg. 2017. Type Systems for Information Flow Control: The Question of Granularity. *ACM SIGLOG News* 4, 1 (2017), 6–21.
- Vineet Rajani and Deepak Garg. 2018. Types for Information Flow Control: Labeling Granularity and Semantic Models. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 233–246.

- Indrajit Roy, Donald E Porter, Michael D Bond, Kathryn S McKinley, and Emmett Witchel. 2009. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 63–74.
- Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. 2020. Lattice-Based Information Flow Control-by-Construction for Security-by-Design. In *FormalISE*. To appear.
- Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *CSF*. IEEE, 186–199.
- Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *J-SAC* 21, 1 (2003), 5–19.
- Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and Principles. *Journal of Computer Security* 17, 5 (2009), 517–548.
- Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick Kourie, and Bruce W Watson. 2018. Towards Confidentiality-by-Construction. In *ISO/IEC JTC1 SC22 WG2 N11244*. Springer, 502–515.
- Vincent Simonet. 2003. Flow Caml in a Nutshell. In *APPSEM-II*. 152–165.
- Alley Stoughton, Andrew Johnson, Samuel Beller, Karishma Chadha, Dennis Chen, Kenneth Foner, and Michael Zhivich. 2014. You Sank My Battleship! A Case Study in Secure Programming. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security (Uppsala, Sweden) (PLAS'14)*. Association for Computing Machinery, New York, NY, USA, 2–14.
- Martin Strecker. 2003. Formal Analysis of an Information Flow Type System for MicroJava. *Technische Universität München, Tech. Rep* (2003).
- Qi Sun, Anindya Banerjee, and David A Naumann. 2004. Modular and Constraint-Based Information Flow Inference for an Object-Oriented Language. In *SAS*, Vol. LNCS. Springer, 84–99.
- Tachio Terauchi and Alex Aiken. 2005. Secure Information Flow as a Safety Problem. In *SAS*, Vol. LNCS. Springer, 352–367.
- Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. 2012. Family-based Deductive Verification of Software Product Lines. In *GPCE*. 11–20.
- Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. 2019. From Fine-to Coarse-Grained Dynamic Information Flow Control and Back. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–31.
- Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *JCS* 4, 2/3 (1996), 167–188.
- Jian Xiang and Stephen Chong. 2021. Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 18–35.
- Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. *Acm Sigplan Notices* 50, 4 (2015), 503–516.
- Lantian Zheng, Stephen Chong, Andrew C Myers, and Steve Zdancewic. 2003. Using Replication and Partitioning to Build Accure Distributed Systems. In *2003 Symposium on Security and Privacy, 2003*. IEEE, 236–250.

## **A.7. Lattice-Based Information Flow Control-by-Construction for Security-by-Design**

# Lattice-Based Information Flow Control-by-Construction for Security-by-Design

Tobias Runge  
TU Braunschweig  
Germany  
tobias.runge@tu-bs.de

Thomas Thüm  
University of Ulm  
Germany  
thomas.thuem@uni-ulm.de

Alexander Knüppel  
TU Braunschweig  
Germany  
a.knueppel@tu-bs.de

Ina Schaefer  
TU Braunschweig  
Germany  
i.schaefer@tu-bs.de

## ABSTRACT

Many software applications contain confidential information, which has to be prevented from leaking through unauthorized access. To enforce confidentiality, there are language-based security mechanisms that rely on information flow control. Typically, these mechanisms work post-hoc by checking whether confidential data is accessed unauthorizedly after the complete program is written. The disadvantage is that incomplete programs cannot be interpreted properly and information flow properties cannot be built in constructively. In this work, we present a methodology to construct programs incrementally using refinement rules to follow a lattice-based information flow policy. In every refinement step, confidentiality and functional correctness of the program is guaranteed, such that insecure programs are prohibited by construction. Our contribution is fourfold. We formalize refinement rules for the constructive information flow control methodology, prove soundness of the refinement rules, show that our approach is at least as expressive as standard language-based mechanisms for information flow, and implement it in a graphical editor called CorC. Our methodology is also usable for integrity properties, which are dual to confidentiality.

## KEYWORDS

correctness-by-construction, information flow control, security-by-design

### ACM Reference Format:

Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. 2020. Lattice-Based Information Flow Control-by-Construction for Security-by-Design. In *8th International Conference on Formal Methods in Software Engineering (FormalISE '20)*, October 7–8, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3372020.3391565>

## 1 INTRODUCTION

Today, customers have a high demand for secure software. An important security property of data is *confidentiality*, which means

---

*FormalISE '20, October 7–8, 2020, Seoul, Republic of Korea*

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *8th International Conference on Formal Methods in Software Engineering (FormalISE '20)*, October 7–8, 2020, Seoul, Republic of Korea, <https://doi.org/10.1145/3372020.3391565>.

that no confidential information is leaked to unauthorized or external systems. Another important property is *integrity* to ensure that critical software is functionally correct and is not influenced by other untrusted software parts. To improve the process of developing secure software, security-by-design techniques have been proposed. These techniques provide guidelines for the overall development process to design and implement secure software. For example, a well-known process is the *Security Development Lifecycle (SDL)* by Microsoft [16]. At implementation level, SDL relies on post-hoc program analysis techniques (i.e., techniques applied after the creation of the program) to ensure confidentiality and integrity [13].

The information flow between variables on source code level is mostly analyzed with language-based static analysis techniques [28]. Such techniques specify security policies to determine the permitted information flow between variables in the program. For example, we may define a policy with two confidentiality levels, high and low, arranged in a lattice where variables are categorized into either of the two. To prevent information leaks, the lattice-based information flow policy prohibits a flow from high to low variables. The same applies for trusted and untrusted variables with a policy that prohibits an information flow from untrusted to trusted variables (i.e., to preserve integrity). As shown by Biba [12], integrity can be seen as a dual to confidentiality, which means that either of them can be checked with the same information flow analysis techniques. Standard information flow analyses are based on security types systems [28, 31]. Such a type system assigns to every variable and expression an explicit security type. A set of typing rules describes the allowed information flow and discards programs that violate the security policy.

In contrast to post-hoc analyses that cannot ensure information flow properties during program construction, but only check programs after their creation, we propose to develop programs that are secure by construction analogous to the correctness-by-construction (CbC) approach for functional correctness [18]. Guided by a pre-/postcondition specification, an abstract program is refined stepwise to a concrete implementation. By applying a sound set of refinement rules, the resulting program is correct by construction. In this paper, we propose Information Flow Control-by-Construction (IFbC) to create functionally correct programs, which also satisfy a lattice-based information flow policy for capturing confidentiality and integrity. The information flow policy can be



specified in any bounded upper semi-lattice (i.e., security levels are arranged in a partially ordered set representing the allowed direction of information flow).

In every step of the program construction in IFbC, the security levels associated with variables are updated according to our refinement rules, and therefore prevent a violation of the information flow policy. Furthermore, the current status of all variables can be observed in (partial) programs after each refinement step. To give programmers more flexibility while constructing a program, we allow to reverse the information flow in appropriate cases. We introduce a *declassify* operation, which can be used if the programmer encrypts or otherwise disguises the confidential information. As the refinement rules also take functional correctness into account, programmers using our methodology create programs that meet two properties, namely functional correctness and security.

In this paper, we demonstrate the strengths of a constructive methodology to develop secure and correct programs. We give two examples to emphasize the advantage of ensuring confidentiality and integrity throughout the development process, rather than having to check this property afterwards. With a sound set of refinement rules, developers can never construct an insecure program, contrary to security type systems that only discard insecure programs. Therefore, IFbC can reduce the post-hoc analysis effort or even make it obsolete, as developers are guided by constructive rules to an already secure program [32]. The IFbC approach contributes to the security-by-design paradigm to close the gap of a constructive process at implementation level. It can be used supplementary to existing processes and analyses for security-critical programs during development.

IFbC presented in this paper extends C14bC by Schaefer et al. [29]. C14bC uses a confidentiality specification with only two levels, high and low, and refinement rules to ensure the confidentiality of programs written in a simple while-language without method calls. Moreover, Schaefer et al. [29] discussed potential tool support for this approach. Finally, we list the four contributions of this work.

- We create an IFbC methodology to construct functionally correct and secure programs regarding a lattice-based confidentiality and integrity policy. Confidentiality, integrity, and functional correctness are ensured simultaneously while constructing the program. The underlying language of IFbC is also extended by method calls to support more meaningful programs.
- We prove the soundness of the proposed refinement rules, such that a program constructed by IFbC never violates our information flow policy.
- We show that IFbC is at least as expressive as a type system for lattice-based information flow control to justify that IFbC can be used supplementary in a program development process.
- We implement the IFbC methodology in a tool called CorC and discuss applicability of our approach.

## 2 FOUNDATIONS

In this section, we provide the background on correctness-by-construction and information flow in order to introduce IFbC in

$\{P\} S \{Q\}$	<i>can be refined to</i>	
<i>Skip</i> :	$\{P\} \text{ skip } \{Q\}$ iff $P$ implies $Q$	(1)
<i>Assignment</i> :	$\{P\} x := E \{Q\}$ iff $P$ implies $Q[x := E]$	(2)
<i>Composition</i> :	$\{P\} S1 ; S2 \{Q\}$ iff there is $M$ such that $\{P\} S1 \{M\}$ and $\{M\} S2 \{Q\}$	(3)
<i>Selection</i> :	$\{P\} \text{ if } G \text{ then } S1 \text{ else } S2 \text{ fi } \{Q\}$ iff $\{P \wedge G\} S1 \{Q\}$ and $\{P \wedge \neg G\} S2 \{Q\}$	(4)
<i>Repetition</i> :	$\{P\} \text{ do } G \rightarrow S \text{ od } \{Q\}$ iff there is an invariant $I$ and a variant $V$ such that ( $P$ implies $I$ ) and ( $I \wedge \neg G$ implies $Q$ ) and $\{I \wedge G\} S \{I\}$ and $\{I \wedge G \wedge V = V_0\} S \{I \wedge 0 \leq V < V_0\}$	(5)
<i>Weaken pre</i> :	$\{P'\} S \{Q\}$ iff $P$ implies $P'$	(6)
<i>Strengthen post</i> :	$\{P\} S \{Q'\}$ iff $Q'$ implies $Q$	(7)
<i>Method call</i> :	$\{P\} M(a_1 \dots a_n) \{Q\}$ for a method $\{P'\} M(z_1 \dots z_n) \{Q'\}$ iff $P = P'[z_i \setminus a_i]$ and $Q = Q'[z_i^{\text{old}}, z_i \setminus a_i^{\text{old}}, a_i]$	(8)

Figure 1: Correctness-by-construction refinement rules [18]

the subsequent section. We also introduce lattices as underlying mathematical structure for lattice-based information flow policies.

### 2.1 Functional Correctness-by-Construction

Correctness-by-construction (CbC) [18] is an approach to construct programs guided by a pre-/postcondition specification. CbC starts with an abstract Hoare triple  $\{P\} S \{Q\}$  consisting of a precondition  $P$ , an abstract statement  $S$ , and a postcondition  $Q$ . This triple is successively refined using a set of refinement rules to a concrete implementation, which satisfies the specification. For simplicity in this paper, we consider the guarded command language introduced by Dijkstra [15]. Each of the refinement rules takes an abstract statement and replaces it with a more concrete guarded command language statement. Every refinement rule preserves the correctness of the program if a discharged side condition is proven correct [25].

The eight considered refinement rules are shown in Fig. 1. As concrete instructions, we have *skip*, *assignment*, and *method call* with call by value-result. *Composition* is used for a sequence of statements. *Selection* and *repetition* are used for the control flow of the program. In Fig. 1, the side conditions for applicability of a refinement rule are shown. For example, when refining to a method call, it has to be proven that the pre-/postcondition specification of the refined triple is equal to the specification of the called method. This refinement rule also requires that the parameters of the method are correctly passed where  $a_i$  are the actual parameters and  $z_i$  are the formal parameters. The parameters with superscript *old* refer to parameters before the execution of the method.

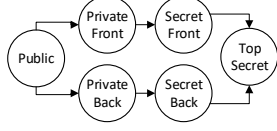


Figure 2: Example of a lattice of confidentiality levels

## 2.2 Information Flow Control

Information flow control mechanisms [28, 31] are used to specify programs with respect to a security policy. The policy can establish confidentiality of processed information to prevent leaks of unauthorized information, or it can guarantee integrity of the processed information by ensuring that trusted parts are not influenced by untrusted parts. Both properties can be analyzed by considering the information flow, as confidentiality can be modeled as dual to integrity [12, 28]. Confidentiality requires that information flow to specific destinations is prevented. Similarly, integrity requires that a flow from specific sources is prevented to ensure that the system is not harmed by untrusted sources. Integrity also requires a functionally correct program because incorrect methods can violate the integrity by computing wrong data. Correctness can be achieved with the presented CbC approach of Section 2.1.

To give an example of a security policy for confidentiality, we consider a company with two different departments. A front office that should know personal information of customers (e.g., their name and age) and a back office that should know critical financial information of customers. The back office does not know other personal information to make unbiased decisions. The front office on the other hand should treat the customers without being influenced by their financial status. To establish this policy, a lattice as in Fig. 2 can be used. This lattice also includes `Public` data for general access, and `Top Secret` data for access by the management. The front and back office are also divided into two levels, `Private` and `Secret`, for data with different confidentiality levels. The arrows in the graph show the allowed flow directions.

*Lattice.* Bell, LaPadula [11], and Denning [14] were the first to arrange confidentiality levels in a lattice. This arrangement of confidentiality levels in our example fulfills the requirements of a bounded upper semi-lattice. A lattice is a structure  $\langle L, \leq, \text{lub}, \top, \perp \rangle$  where  $L$  is a set of levels and  $\leq$  is a partial order (e.g., `Public`  $\leq$  `Private Front`). The relation operator is reflexive, antisymmetric, and transitive, but per definition not every pair of elements need to be comparable. An upper bound in the lattice is defined as follows: for a set of elements  $X \subseteq L$ , an upper bound  $y$  exists if  $\forall x \in X : x \leq y$ . The element  $u$  is the least upper bound ( $\text{lub} : \mathcal{P}(X) \rightarrow X$ ), of all  $x \in X$  if  $u \leq y$  for all upper bounds  $y$ . We restrict the lattice to be a bounded upper semi-lattice, which has the greatest element  $\top$  and the least element  $\perp$ , (i.e.,  $\perp \leq a \leq \top$  for every  $a \in L$ ). For every combination of levels, a unique least upper bound ( $\text{lub}$ ) must exist. The  $\text{lub}$  is used to calculate the least security level such that violations of the information flow policy are prevented (e.g., that no financial information flows to a member of the front office).

We distinguish between two information flow types. Information can flow *directly* through an assignment statement, which assigns

$$\begin{array}{lll}
 (1) \vdash x : \tau \text{ var} & (2) \frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'} & (3) \frac{\vdash p : \rho \quad \vdash \rho \subseteq \rho'}{\vdash p : \rho'} \\
 (4) \frac{\vdash \tau \subseteq \tau'}{\vdash \tau' \text{ cmd} \subseteq \tau \text{ cmd}} & (5) \frac{\vdash x : \tau \text{ var} \quad \vdash e' : \tau}{\vdash x := e' : \tau \text{ cmd}} & \\
 (6) \frac{\vdash c : \tau \text{ cmd} \quad \vdash c' : \tau \text{ cmd}}{\vdash c; c' : \tau \text{ cmd}} & (7) \frac{\vdash e : \tau \quad \vdash c : \tau \text{ cmd}}{\vdash \text{while } e \text{ do } c : \tau \text{ cmd}} & \\
 (8) \frac{\vdash e : \tau \quad \vdash c : \tau \text{ cmd} \quad \vdash c' : \tau \text{ cmd}}{\vdash \text{if } e \text{ then } c \text{ else } c' : \tau \text{ cmd}} & & 
 \end{array}$$

Figure 3: Security type system [31]

data to another variable. Here, we have to ensure that the assigned variable gets a security level of at least the *lub* of all variables used in the expression to prevent a leak. Information can also flow *indirectly* through conditional or loop statements. If confidential data is used in a guard of a conditional statement, the chosen branch gives information about the variables in the guard. Therefore, the confidentiality level in the branches must be the least upper bound of the levels in the guards, too.

*Security Type System.* A security type system [28] ensures the compliance of a program with an information flow policy. A set of typing rules determine the allowed information flow and discard programs, which violate the security policy. An excerpt of a type system by Volpano et al. [31] is shown in Fig. 3. Here, we have security levels  $\tau$  that are arranged in a lattice  $L \langle L, \leq \rangle$  with  $\tau \in L$ . The language consists of statements  $c$  that are typed with  $\tau \text{ cmd}$ , expressions  $e$  that are typed with a security level  $\tau$ , and variables  $x$  that are typed with  $\tau \text{ var}$  (Rule 1). The typing of expressions should prevent a leak through direct information flow, and the typing of statements is used for the indirect information flow. Variables are expressions. Expressions and statements are both phrases  $p$ . The different types  $\tau \text{ cmd}$ ,  $\tau \text{ var}$ , and  $\tau$  are all phrase types  $\rho$ . The partial order of confidentiality levels ( $\leq$ ) is extended to a subtype relation  $\subseteq$  (rules 2–4) to use subtyping in the other typing rules 5–8. Typing Rule 5 shows a secure assignment. To assign expression  $e'$  to  $x$ , both expressions must agree on their security level. Through subtyping (2–4), an assignment from a lower to a higher security level is allowed. The rules 6–8 describe the standard program flow constructs for sequence, conditional, and repetition. Here, the security levels of the commands  $c$ ,  $c'$ , and guards  $e$  have to be equal or subtyping has to be used.

## 3 INFORMATION FLOW CONTROL-BY-CONSTRUCTION

To motivate the IFbC approach, we give two examples. The first example creates a confidential program, and the second example uses an information flow policy to ensure integrity of a program.

*Auction Example for Confidentiality.* To illustrate IFbC for confidentiality, we construct a program for an auction. The input is an array of bids for an item, and the goal is to find the maximum bid that wins the auction. The array of bids is traversed to find this maximum, which is published. We assume three security levels `public`, `private`, and `secret` with `public`  $<$  `private`  $<$  `secret` and each variable is labeled with one of these security levels.

```

1  pre: publishBid = 0
2  post: \forall int x; ((x >= 0
3        & x < bids.length)
4        -> (publishBid >= bids[x]))
5  void auction(private int[] bids,
6              public int publishBid) {
7      public int i = 0;
8      secret int highestBid = 0;
9      do (i < bids.length) {
10         if (highestBid < bids[i]) {
11             highestBid = bids[i];
12         } else {
13             skip
14         } fi
15         i = i + 1;
16     } od
17     publishBid = declassify(highestBid);
18 }

```

Listing 1: Program of the auction example

The auction method is specified such that it gets as input a private array `bids` and a public variable `publishBid` (`pB`). The method sets `pB` to the maximum bid of the auction. In IFbC, parameters are passed by value-result. The security levels of other local variables used in the code are not specified yet. If needed, programmers can add additional variables with an initially chosen security level while constructing the program, where the resulting security level of the variables can be changed in the program to prevent leaks. Additionally, a functional specification of the program can be given to construct a functionally correct program. The refinement rules of Fig. 1 are used to guarantee the functional correctness. Simultaneously, refinement rules of IFbC are used to ensure the specified confidentiality policy.

To construct the program with IFbC, we start with a provided IFbC triple  $\{\mathcal{V}^{pre}, P\} S \{\mathcal{V}^{post}, Q\}[\eta]$ . This specification indicates the security levels of variables before (labeling function  $\mathcal{V}^{pre}$ ) and after ( $\mathcal{V}^{post}$ ) program execution. An instance would be the specification of the auction problem as above:  $\mathcal{V}^{pre}, \mathcal{V}^{post} := \text{public } pB, \text{ private } bids$ . The security context  $\eta$  is used to reason about indirect information flow. It tracks the security level of guards used in conditional or loop statements. Furthermore, the triple includes the abstract statement  $S$  that is refined to a concrete program. The functional specification is provided as logical precondition  $P$  and postcondition  $Q$  (cf. pre and post in Listing 1). In the following, we construct the program and refer to the functional refinement rules that are applied. By refining the program, we can also guarantee that the security specification is met by construction.

In Fig. 4, we show the refinement steps for the auction example in a graphical notation. Here, we omit the functional specification to focus on the information flow. The postcondition contains the public variable `publishBid` (indicated by the predicate `public(pB)` in the graphic), the private variables `bids` and `i` (a control variable of the loop), and the secret variable `highestBid` (`hB`) (a temporary variable for the maximum bid). The precondition specifies that `publishBid` is public and `bids` has a private security level. The additional variables `i` and `hB`, which do not occur in the specification above, are added by the programmer while constructing

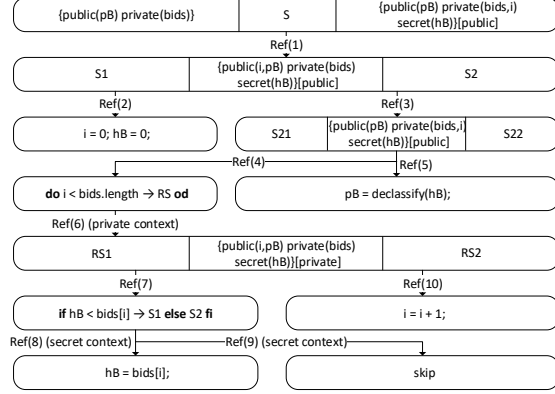


Figure 4: Refinement steps for the auction example

the program. Their resulting confidentiality levels are determined through the application of the refinement rules.

To construct the algorithm as in Listing 1, we want to divide the problem into three parts, an initialization of some temporary variables, the loop through the array of `bids` to search for the highest bid, and the assignment of the highest bid to the public variable `pB`. The first split into the initialization and the rest of the program is done with Refinement (1). It introduces a composition statement (cf. Rule 3 in Fig. 1), splitting the problem in two abstract subproblems  $S1$  and  $S2$  with an intermediate specification, which is calculated by IFbC while refining the program. The intermediate specification presents the security level of variables between statements.

In Refinement (2), the initialization of the temporary variables is done with the assignment statement `i = 0; hB = 0;` (cf. Rule 2, indeed it is a multi-step refinement with Rule 3 and 2). The statement initializes `i` as public and `highestBid` as secret variable, as declared by the programmer. In the declarations, the variables are initially labeled by the programmer, and further refinement rules ensure that these labeled variables are correctly adjusted during the refinements. In the postcondition of statement  $S2$ , which is the postcondition of the starting triple, the variable `i` has a private security level. Here, we can see that the security level of `i` is updated from public to private in the program to prevent a leak.

In Refinement (3), we further split the second part of the program with the composition rule to iterate through the array of `bids` first, and then, to assign the highest bid to a public variable with the use of a declassify operation. Refinement (5) assigns the highest bid and is explained after the refinement of the loop.

Refinement (4) creates the loop to iterate through the array of `bids` searching for the maximum as long as the control variable `i` is smaller than the length of the array (cf. Rule 5 for functional correctness). As the variable `bids` in the guard of the loop (`i < bids.length`) has a private security level, the security context of the loop body has to be increased to private to prevent leaks through indirect information flow. That means, sub-statements of the repetition can only assign information to variables of at least the security level of the new security context. For example, if we would assign to a public variable in the loop body, an attacker

could deduce that the guard was evaluated to true by reading that public variable (e.g., `bids.length` is bigger than `i`).

The refinements (6)–(10) create the loop body which compares the next element of the array with the current highest bid. If the next element is greater, we update the highest bid. Refinement (6) splits the loop body with a composition into a check of the next bid and the increment of the loop variable. The refinements (7)–(9) establish the selection to check whether the next bid is higher than the current highest bid. As `hb` is used in the guards, the security context inside the selection statement is increased to secret. In Refinement (8), we assign a new highest bid to our variable (`hb = bids[i]`). As `hb` is already secret, the security level stays the same, otherwise the security level of the variable has to be increased to secret because of the secret security context. In the case that the next bid is smaller or equal to the highest bid, Refinement (9) introduces a skip statement to not alter the program state.

The assignment in Refinement (10) increments the loop counter. Here, we increase the security level of `i` through the private security context. The security level of the public variable `i` is set to private. This increase of the security level propagates through the program, and therefore the security level of `i` is private in the initial triple of the program.

In Refinement (5), we construct the last part of the program, the assignment to the variable `pb`. Normally, by assigning secret data to a public variable, the public security level has to be increased to secret. This prevents a leak through direct information flow, as secret data would be accessible through a public variable. With a declassify operation, programmers can prevent the increase of the security level (e.g., if they are sure that the confidential data is allowed to be published, or the data is encrypted beforehand).

*Banking Example for Integrity.* In Fig. 5, we show a second example demonstrating how IFbC works for integrity. A user withdraws money from a bank account and the balance should be updated if the withdrawal is trustworthy. In the other case, the balance is not updated to secure the integrity of the bank. The precondition of the program specifies that it gets two trusted variables `balance` and `checked` (`checked` is used as parameter to return the result of the method), and an untrusted variable `withdraw` as input. The postcondition specifies that these security levels must not be altered. The allowed flow is from trusted to untrusted. The complete program with a functional specification is shown in Listing 2. The balance is reduced by the value of `withdraw` if the value of variable `checked` is true. In the other case, the balance is not altered. The Boolean variable `checked` is set by a method call to `check`.

To construct the program, we use a composition Refinement (1) to split the problem into a check whether the withdrawal is allowed and the update of the `balance` variable. Refinement (2) introduces a method call to check whether the system can trust the input variable `withdraw`. If this is the case, the variable `checked` is set to a true value. When calling the method, all parameters are passed by value-result, and therefore their security level can be changed. For example, the method `check` could be specified that it returns `balance` with an untrusted security level, as we allow an update of the security levels from trusted to untrusted. Then, the bank method would have to proceed with an untrusted variable. In our case, the security levels stay the same because we assume that

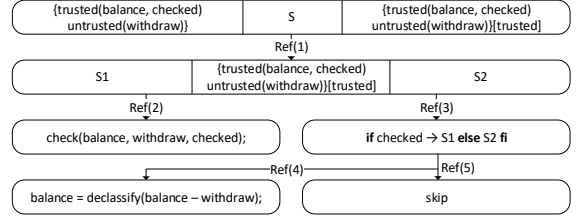


Figure 5: Refinement steps for the banking example

```

1  pre: true
2  post: (!checked -> balance ==
3        \old(balance) & (checked -> balance
4         == \old(balance) - withdraw);
5  void bank(trusted int balance,
6            trusted boolean checked,
7            untrusted int withdraw) {
8    check(balance, withdraw, checked);
9    if (checked) {
10     balance = declassify(balance - withdraw);
11   } else {
12     skip
13   } fi
14 }

```

Listing 2: Program of the banking example

the method `check` is specified that way. To verify that the method `check` fulfills its specification, it would be created with IFbC.

Refinements (3)–(5) introduce the selection statement to set the new balance of the bank account. As a trusted variable is used in the guard, the security context stays the same. With the declassify operation, we can calculate the new balance in the then-branch. Without declassify, it is not permitted to assign an untrusted value to the trusted variable `balance`. In the else-branch, a skip statement is used that does not alter the program.

For clarity, we give individual examples for confidentiality and integrity, but both policies can be ensured in the same program simultaneously by construction, as the IFbC refinement rules operate on any lattice of security levels. Practically, the variables would be labeled with a confidentiality and an integrity level, which are updated individually. Another possibility is to create the power set of both lattices and label every variable with a combination of security levels [31].

## 4 FORMALIZING INFORMATION FLOW CONTROL-BY-CONSTRUCTION

In this section, we formalize IFbC for the construction of functionally correct and secure programs. With this approach, programmers can incrementally construct programs, where the security levels are organized in a lattice structure to guarantee a variety of confidentiality and integrity policies. IFbC defines seven refinement rules to create secure programs. As these rules are based on refinement rules for correctness-by-construction, programmers can create programs that are functionally correct and secure.

$Vars$	Set of program variables
$S$	Statement (from the GCL [15])
$x \in Vars$	Program variable
$E$	Expressions over the program variables in $Vars$
$vars(E) \subseteq Vars$	Set of variables occurring in expression $E$
$L$	Bounded upper semi-lattice ( $L, \leq$ ) of security levels
$\mathcal{V}^{pre}, \mathcal{V}^{post}, l : Vars \rightarrow L$	Labeling function to map a variable to a security level
$lub_L : \mathcal{P}(L) \rightarrow L$	Least upper bound of the security levels in $L$
$\eta \in L$	Security context
$\{\mathcal{V}^{pre}, P\} S \{\mathcal{V}^{post}, Q\} [\eta]$	IFbC triple

Figure 6: Basic notations for IFbC

#### 4.1 Refinement Rules for Program Construction

To formalize the IFbC rules, we introduce in Fig. 6 basic notations for variables and security levels, which are used in the refinement rules. Every variable of the program is associated to one security level. Levels are arranged in a bounded upper semi-lattice with one greatest and one least level. The functional and security specification of a program is defined by an IFbC triple  $\{\mathcal{V}^{pre}, P\} S \{\mathcal{V}^{post}, Q\} [\eta]$ . As a Hoare triple, the IFbC triple consists of a precondition  $\{\mathcal{V}^{pre}, P\}$ , an abstract statement  $S$ , and a postcondition  $\{\mathcal{V}^{post}, Q\}$ . The functional specification is declared in the logical formulas  $P$  and  $Q$ . In the following, we focus on security, so the functional specification will be omitted. The labeling function  $\mathcal{V}^{pre}$  assigns security levels to all variables before the statement  $S$  is executed and  $\mathcal{V}^{post}$  assigns security levels to all variables after the execution. The label  $\eta$  is used to capture the security context of the IFbC triple. This security context is used to reason about implicit information flow by tracking the security levels of guards in conditional or loop statements. The refinement rules replace an abstract statement by a more concrete statement. In the refined triple, the security levels of the variables are updated to implement the security policy of the program.

**Skip.** The first IFbC rule introduces a skip statement, which does not alter the program. It refines an IFbC triple  $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\} [\eta]$  to a skip. The rule is applicable if the variables and their associated security levels stay the same.

**RULE 1 (SKIP).**  
 $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\} [\eta]$  is refinable to  $\{\mathcal{V}^{pre}\} \text{skip} \{\mathcal{V}^{post}\} [\eta]$  iff  $\mathcal{V}^{post}(x) = \mathcal{V}^{pre}(x)$  for all  $x \in Vars$ .

**Assignment.** With the assignment rule, an abstract statement  $S$  is refined to an assignment of the form  $x := E$ . This represents explicit information flow from the variables in the expression  $E$  to the variable  $x$  on the left-hand side. This direct flow can cause a leak if data with a higher security level is assigned to  $x$ . We can prevent this leak by enforcing the security level of  $x$ .

To apply the refinement, the labeling function  $\mathcal{V}^{post}$  has to be altered. The new security level of the variable  $x$  is determined by the least upper bound of the security levels of all variables in

the expression, the security level of the context to consider the indirect information flow and the security level of  $x$  itself. This new security level is assigned to  $x$  in the labeling function  $\mathcal{V}^{post}$  in the postcondition of the IFbC triple. So, the only difference of  $\mathcal{V}^{pre}$  and  $\mathcal{V}^{post}$  is the update of the security level of variable  $x$ .

**RULE 2 (ASSIGNMENT).**  
 $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\} [\eta]$  is refinable to  $\{\mathcal{V}^{pre}\} x := E \{\mathcal{V}^{post}\} [\eta]$  iff  $\mathcal{V}^{post}(y) = \mathcal{V}^{pre}(y)$  for all  $y \in Vars \setminus \{x\}$ , and  $\mathcal{V}^{post}(x) = lub(\{\mathcal{V}^{pre}(v) \mid v \in vars(E)\} \cup \{\mathcal{V}^{pre}(x), \eta\})$ .

**Composition.** With the composition rule, an abstract IFbC triple  $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\} [\eta]$  is refined to two triples  $\{\mathcal{V}^{pre}\} S1 \{\mathcal{V}'\} [\eta]$  and  $\{\mathcal{V}'\} S2 \{\mathcal{V}^{post}\} [\eta]$ , which are executed sequentially. Both triples can be further refined. To apply the rule, a labeling function  $\mathcal{V}'$  is introduced, which assigns a security level to all program variables after the execution of the first statement and before the execution of the second statement. The exact labeling function  $\mathcal{V}'$  is determined by refining  $S1$  and  $S2$  to concrete statements. The labeling functions  $\mathcal{V}^{pre}$  and  $\mathcal{V}^{post}$  and the security context  $\eta$  are not changed. For all variables, the security level can only be increased by the program. To capture a reverse information flow in specific cases, the declassify operation and new variables are used.

**RULE 3 (COMPOSITION).**  
 $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\} [\eta]$  is refinable to  $\{\mathcal{V}^{pre}\} S1; S2 \{\mathcal{V}^{post}\} [\eta]$  iff there exists a labeling function  $\mathcal{V}' : Vars \rightarrow L$  such that  $\{\mathcal{V}^{pre}\} S1 \{\mathcal{V}'\} [\eta]$  and  $\{\mathcal{V}'\} S2 \{\mathcal{V}^{post}\} [\eta]$  and for all  $v \in Vars : \mathcal{V}^{pre}(v) \leq \mathcal{V}'(v) \leq \mathcal{V}^{post}(v)$ .

**Selection.** The selection rule refines an abstract statement  $S$  to an if statement  $\text{if}(G) \rightarrow S1 \text{ else } S2 \text{ fi}$ . Here, an implicit leak can occur as the selected branch reveals information about the guard. To prevent this, the statements in the branches have to be labeled with at least the security level of the guard. As selection statements can be nested, a security context is used to track the current security level that is needed to prevent an implicit leak.

By applying the refinement rule, the security context of the sub-statements have to be adjusted to the least upper bound of the security level of the if-guard and the security context of the outer selection statement. Both sub-statements with the new security context can be further refined.

**RULE 4 (SELECTION).**  
 $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\} [\eta]$  is refinable to  $\{\mathcal{V}^{pre}\} \text{if } G \rightarrow S1 \text{ else } S2 \text{ fi} \{\mathcal{V}^{post}\} [\eta]$  iff  $\{\mathcal{V}^{pre}\} S1 \{\mathcal{V}^{post}\} [\eta']$  and  $\{\mathcal{V}^{pre}\} S2 \{\mathcal{V}^{post}\} [\eta']$  with  $\eta' = lub(\{\mathcal{V}^{pre}(v) \mid v \in vars(G)\} \cup \{\eta\})$ .

**Repetition.** The repetition rule introduces a classic while loop. By executing the loop, information about the guard is revealed. To prohibit this indirect leak, the security context of the inner loop statement is adjusted to the least upper bound of the security levels of the loop-guard and the security context of the outer repetition statement.

**RULE 5 (REPETITION).**  
 $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\} [\eta]$  is refinable to  $\{\mathcal{V}^{pre}\} \text{do } G \rightarrow S1 \text{ od} \{\mathcal{V}^{post}\} [\eta]$  iff  $\{\mathcal{V}^{pre}\} S1 \{\mathcal{V}^{post}\} [\eta']$  with  $\eta' = lub(\{\mathcal{V}^{pre}(v) \mid v \in vars(G)\} \cup \{\eta\})$

## 4.2 Method Call Rule

In a method call, all variables are passed by value-result and appear in the specification of the method. The security level of these passed variables may change, while the security level of other variables remains the same. By calling the method, the parameters of the caller are assigned to the parameters of the called method and the reverse assignment is done when returning from the method. It has to be ensured that in the beginning the security levels of variables of the called method are higher than or equal to the security levels of variables of the caller to prevent flows from higher to lower security levels. It also has to be ensured that the security levels in the postcondition of the caller are higher than or equal to the security levels of the called method for the same reason. For example, a secure value of the method has to be assigned to a variable with at least this security level in the program of the caller. Additionally, the called method has to satisfy its specification, which can be shown in a separate IFbC refinement.

**RULE 6 (METHOD CALL).**  
 $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\}[\eta]$  is refinable to  $\{\mathcal{V}^{pre}\} M(a_1, \dots, a_n) \{\mathcal{V}^{post}\}[\eta]$  iff for a method  $\{\mathcal{V}^{pre}_{call}\} M(z_1, \dots, z_n) \{\mathcal{V}^{post}_{call}\}[\eta]$  and for all parameters:  $\mathcal{V}^{pre}(a_i) \leq \mathcal{V}^{pre}_{call}(z_i) \wedge \mathcal{V}^{post}_{call}(z_i) \leq \mathcal{V}^{post}(a_i)$  where  $a_i$  are the actual parameters and  $z_i$  are the formal parameters.

## 4.3 Declassification

With our information flow policy, we are not allowed to assign an expression with a higher security level to a variable with a lower security level without increasing the security level of the variable. This restricts the possibility to develop meaningful programs; in some cases the information flow from a more secure variable to a less secure one should be possible. For example, if a password is saved into a secure variable, an encrypted or hashed version of the password should be assignable to a less confidential variable. A declassification operator [22, 33] can be used to allow the assignment, but it should only be used if the programmer is sure that no secure information is leaked.

The declassification rule is a specialized assignment rule, where an expression  $E$  assigned to variable  $x$  is surrounded by the `declassify` operator. With this rule, the security level of the assigned variable is only set to the least upper bound of its security level and the security context. The difference to the standard assignment rule is that the security levels of variables of the assigned expression are not used to determine the new security level. The declassification refinement rule is only meaningful if the assigned expression would increase the security level of the assigned variable. If the security levels of all variables of the expression are lower than the security level of the assigned variable, the standard assignment rule and the declassification rule behave the same.

**RULE 7 (DECLASSIFICATION ASSIGNMENT).**  
 $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\}[\eta]$  is refinable to  $\{\mathcal{V}^{pre}\} x := \text{declassify}(E) \{\mathcal{V}^{post}\}[\eta]$  iff  $\mathcal{V}^{post}(y) = \mathcal{V}^{pre}(y)$  for all  $y \in \text{Vars} \setminus \{x\}$ , and  $\mathcal{V}^{post}(x) = \text{lub}(\{\mathcal{V}^{pre}(x), \eta\})$ .

## 5 PROOF OF SOUNDNESS AND EXPRESSIVENESS OF IFbC

We want to ensure that programs constructed with IFbC are secure. We assume that `declassify` is correctly used by the programmer because IFbC can detect the use of `declassify`, but it can not prevent an inappropriate application. In the following, we prove soundness of our IFbC rules.

**DEFINITION 1 (SECURE PROGRAM).**

Let  $S$  be an IFbC program and  $\{\mathcal{V}^{pre}\} x := E \{\mathcal{V}^{post}\}[\eta]$  be an arbitrary IFbC triple in program  $S$ . Moreover, let  $G$  be the (possibly empty) set of all defined guards along the refinements from the root to that triple (i.e., in conditional statements and loops). We say that program  $S$  is secure (denoted by `secure(S)`) iff for all such triples the following two conditions hold:

- $\forall v \in \text{vars}(E) : \mathcal{V}^{post}(x) \geq \mathcal{V}^{pre}(v)$  (No direct leak)
- $\forall v \in \text{vars}(G) : \mathcal{V}^{post}(x) \geq \mathcal{V}^{pre}(v)$  (No indirect leak)

The variable  $x$  must have a security level that is greater than or equal to all security levels of variables that are in the expression  $E$  to prevent an assignment of secure information to an insecure variable. Indirect information flow leaks are prevented if no information can be deduced by analyzing the guards of conditional statements or loops. The variable  $x$  must have at least the security level of all guards used in the refinement branch.

To verify the soundness of IFbC, we start with a lemma to reason about indirect information flow. By assigning an expression  $E$  to a variable  $x$ , we know that  $x$  has at least the security level of the security context  $\eta$  that captures the current security level to prevent indirect leaks (i.e.,  $\eta$  is used to track the security levels of guards used in the refinement branch).

**LEMMA 1 (CONFINEMENT).**

Let  $\{\mathcal{V}^{pre}\} x := E \{\mathcal{V}^{post}\}[\eta]$  be an IFbC triple, then  $\mathcal{V}^{post}(x) \geq \eta$ .

**PROOF.** Confinement is proven by the definition of the refinement Rule 2 (Assignment). The new security level of  $x$  is computed by  $\mathcal{V}^{post}(x) = \text{lub}(\{\mathcal{V}^{pre}(v) \mid v \in \text{vars}(E)\} \cup \{\mathcal{V}^{pre}(x), \eta\})$ , and therefore  $\mathcal{V}^{post}(x) \geq \eta$  by the definition of the least upper bound.  $\square$

**Soundness.** With this lemma, we can prove the soundness theorem, which states that a program is secure, if it is constructed using our refinement rules.

**THEOREM 1 (SOUNDNESS).**

If an IFbC triple  $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\}[\eta]$  is refined to  $\{\mathcal{V}^{pre}\} C \{\mathcal{V}^{post}\}[\eta]$  with the IFbC refinement rules without `declassify`, and  $C$  is a concrete program, then `secure(C)` holds.

**PROOF.** We prove the soundness with structural induction. *Skip*, *assignment*, and *method call* are the basis steps because they are the leaves of the refinement tree, and *selection*, *repetition*, and *composition* are proven in the induction step.

**Induction Base:**

- **Assignment:**  $\{\mathcal{V}^{pre}\} S \{\mathcal{V}^{post}\}[\eta]$  is refined to  $\{\mathcal{V}^{pre}\} x := E \{\mathcal{V}^{post}\}[\eta]$ . By using the assignment rule, the new security level of  $x$  is  $\mathcal{V}^{post}(x) = \text{lub}(\{\mathcal{V}^{pre}(v) \mid v \in \text{vars}(E)\} \cup \{\mathcal{V}^{pre}(x), \eta\})$ . We have to show the absence of direct and indirect information flow leaks.

- Case direct information flow: We have to ensure that  $\forall v \in \text{vars}(E) : \mathcal{V}^{\text{post}}(x) \geq \mathcal{V}^{\text{pre}}(v)$ . The assignment rule sets the security level of  $x$  at least to  $\text{lub}(\{\mathcal{V}^{\text{pre}}(v) \mid v \in \text{vars}(E)\})$ . With the definition of  $\text{lub}$ , we know that  $\forall v \in \text{vars}(E) : \mathcal{V}^{\text{post}}(x) \geq \mathcal{V}^{\text{pre}}(v)$ , and therefore no leaks can occur.
- Case indirect information flow: We have to ensure that  $\forall v \in \text{vars}(G)$  of guards  $G$  in the refinement branch:  $\mathcal{V}^{\text{post}}(x) \geq \mathcal{V}^{\text{pre}}(v)$ . As we are at the start of the induction no refinement rule is used so far and no guards  $G$  exist. Using Lemma 1, we know that  $\mathcal{V}^{\text{post}}(x) \geq \eta$ , so no leaks can occur.
- Skip:  $\{\mathcal{V}^{\text{pre}}\} S \{\mathcal{V}^{\text{post}}\}[\eta]$  is refined to  $\{\mathcal{V}^{\text{pre}}\} \text{skip} \{\mathcal{V}^{\text{post}}\}[\eta]$ . As the skip statement has no assignment to a variable, no direct or indirect information flow can exist.
- Method Call: Given a method  $\{\mathcal{V}^{\text{pre}}\} M(z_1, \dots, z_n) \{\mathcal{V}^{\text{post}}\}[\eta]$ , the method call rule assigns the parameters  $z_i$  to the actual parameters and ensures that the security levels are only increased. Therefore, with the assumption that the method itself satisfies its IFbC triple, the method call does not violate the security policy.

**Induction Hypothesis:** For each IFbC triple  $\{\mathcal{V}^{\text{pre}}\} T \{\mathcal{V}^{\text{post}}\}[\eta]$  that was created in  $n$  refinement steps from an abstract IFbC triple  $\{\mathcal{V}^{\text{pre}}\} S \{\mathcal{V}^{\text{post}}\}[\eta]$  (denoted as  $T = \text{refined}(S)$ ),  $\text{secure}(T)$  holds.

**Induction Step:**

- Repetition:  $\{\mathcal{V}^{\text{pre}}\} S \{\mathcal{V}^{\text{post}}\}[\eta]$  is refined to  $\{\mathcal{V}^{\text{pre}}\} \text{do } G \rightarrow S1 \text{ od } \{\mathcal{V}^{\text{post}}\}[\eta]$  with  $\{\mathcal{V}^{\text{pre}}\} S1 \{\mathcal{V}^{\text{post}}\}[\eta']$ . By using the repetition rule, the security context for the statement  $S1$  is set to  $\eta' = \text{lub}(\{\mathcal{V}^{\text{pre}}(v) \mid v \in \text{vars}(G)\} \cup \{\eta\})$ . By using the induction hypothesis, we know that  $\text{secure}(S1)$  holds before introducing the loop. We have to show that the refinement preserves security.
  - Case direct information flow: Since the repetition statement does not introduce an assignment, no direct leak can occur.
  - Case indirect information flow: The repetition statement introduces a guard  $G$ . To prevent an indirect leak, each assigned variable  $x$  in the refinement branch of  $S1$  needs at least the security level of  $G$  ( $\forall v \in \text{vars}(G) : \mathcal{V}^{\text{post}}(x) \geq \mathcal{V}^{\text{pre}}(v)$ ). Therefore, the repetition rule sets the security context from  $\eta$  to  $\eta'$  as shown above, where  $\eta'$  is greater than or equal to every security level of variables in the guard  $G$ . With the correctly updated security context  $\eta'$  and the Confinement Lemma 1 ( $\mathcal{V}^{\text{post}}(x) \geq \eta'$ ), we know that every assignment in the refinement tree of  $S1$  has at least the security level of  $\eta'$ , and therefore the complete program with the repetition statement has no leaks.
- Selection: Selection is similar to repetition. A new guard is introduced and the security context is correctly adjusted. The difference is that the adjusted security context applies for two sub-statements.
- Composition:  $\{\mathcal{V}^{\text{pre}}\} S \{\mathcal{V}^{\text{post}}\}[\eta]$  is refined to  $\{\mathcal{V}^{\text{pre}}\} S1; S2 \{\mathcal{V}^{\text{post}}\}[\eta]$  with an intermediate labeling function  $\mathcal{V}'$ . From the induction hypothesis we know that both triples  $\{\mathcal{V}^{\text{pre}}\} S1 \{\mathcal{V}'\}[\eta]$  and  $\{\mathcal{V}'\} S2 \{\mathcal{V}^{\text{post}}\}[\eta]$  are secure. Since the following applies for all  $v \in \text{Vars} : \mathcal{V}^{\text{pre}}(v) \leq \mathcal{V}'(v) \leq \mathcal{V}^{\text{post}}(v)$ , the security levels can only be increased. No assignment or guard is introduced, so no new direct or indirect leak can occur and the rest of the program is secure through the induction hypothesis. Therefore, we can deduce that  $\text{secure}(S)$  holds.  $\square$

*Expressiveness.* We prove that IFbC is at least as expressive as the information flow type system by Volpano et al. [31]. The type system was already introduced in Section 2.2. Now, we prove that every program, which is type safe (denoted as  $\text{typesafe}(C)$ ), can also be constructed using IFbC. Note that the statement  $c$  of the typing rules and our statements  $S$  are analogous constructs for abstract statements. The security context  $\eta$  of our IFbC approach is also analogous to the type  $\tau \text{ cmd}$  of the statement in the type system.

**THEOREM 2 (EXPRESSIVENESS).**

*For all programs  $C$ , if  $\text{typesafe}(C)$  holds, then there exists  $\{\mathcal{V}^{\text{pre}}\} S \{\mathcal{V}^{\text{post}}\}[\eta]$  as a starting IFbC triple which is refined to the same program  $C$  ( $\text{refined}(S) = C$ ) and  $\text{secure}(C)$  holds.*

**PROOF.** We prove the expressiveness with structural induction on the type derivation. The typing rule for assignments (cf. typing Rule 5 in Fig. 3) is the typing rule for the start of the induction and typing rules 6, 7, and 8 are proven in the induction step.

**Induction Base:**

- Assignment: If  $x := e'$  is type safe, where  $x$  is of type  $\tau \text{ var}$  and  $e'$  is of type  $\tau$ , then we can refine a triple  $\{\mathcal{V}^{\text{pre}}\} S \{\mathcal{V}^{\text{post}}\}[\eta]$  to  $\{\mathcal{V}^{\text{pre}}\} x := e' \{\mathcal{V}^{\text{post}}\}[\eta]$ , where  $x$  and  $e'$  have the same security levels. Subtyping is allowed through typing Rule 2, which is analogous to our lattice-based definition of  $\text{lub}$ .

**Induction Hypothesis:** For each type safe program  $C$  that was typed by  $n$  typing rules, the following holds:  $C = \text{refined}(S)$  and  $\text{secure}(C)$ .

**Induction Step:**

- Typing Rule 6: The rule ensures that if  $C$  and  $C'$  are type safe,  $C; C'$  is also type safe. With the induction hypothesis, we know that  $C$  and  $C'$  are type safe and also the triples  $\{\mathcal{V}^{\text{pre}}\} C \{\mathcal{V}'\}[\eta]$  and  $\{\mathcal{V}'\} C' \{\mathcal{V}^{\text{post}}\}[\eta]$  are secure. By using the composition refinement rule, also  $\{\mathcal{V}^{\text{pre}}\} C; C' \{\mathcal{V}^{\text{post}}\}[\eta]$  is secure as the refinement rule ensures that  $\mathcal{V}'$  is the same in both triples.
- Typing Rule 7: The rule ensures that if  $C$  and  $e$  are type safe,  $(\text{while } e \text{ do } C : \tau \text{ cmd})$  is also type safe. With the induction hypothesis, we know that  $C$  is type safe and also the triple  $\{\mathcal{V}^{\text{pre}}\} C \{\mathcal{V}^{\text{post}}\}[\eta]$  is secure. To prove that the triple  $\{\mathcal{V}^{\text{pre}}\} \text{do } G \rightarrow C \text{ od } \{\mathcal{V}^{\text{post}}\}[\eta']$  is secure, we review the adjustment of the security context in the type system and in our approach. The type  $\tau \text{ cmd}$  of the statement  $C$  can have any type that is more secure than the type of  $e$  (cf. typing rules 4 and 7). This relation is analogously ensured by our repetition rule, which sets the security context  $\eta$  to  $\text{lub}(\{\mathcal{V}^{\text{pre}}(v) \mid v \in \text{vars}(G)\} \cup \{\eta'\})$ . The security level of the context has at least the security level of the guard.
- Typing Rule 8: This rule is similar to repetition. The only difference is that this rule needs two sub-statements  $C$  and  $C'$ .  $\square$

## 6 TOOL SUPPORT AND APPLICATIONS

Instead of proving post-hoc that a program is secure, we create with IFbC programs that are secure by construction. IFbC is at least as expressive as standard type systems and security is guaranteed through the sound set of refinement rules. In order to make IFbC amenable for programmers, we implemented tool support. We discuss the applicability of IFbC at the end.



*Tool Support.* We implemented tool support for IFbC, so that programmers can construct programs, which follow a lattice-based information flow policy. The tool guides a programmer to a secure program with the help of the IFbC refinement rules. In every step of the program refinement, a violation of the information flow policy is prevented by updating the security levels of variables. Simultaneously, refinement rules for correctness-by-construction guarantee the functional correctness of the program.

IFbC is implemented in a graphical editor CorC.<sup>1</sup> The editor is implemented in Java as an Eclipse modeling project. By tracking variables and their security levels, programs can be constructed that are secure with respect to the information flow policy. CorC represents the refinement hierarchy of an IFbC program in a tree structure. Every node represents an IFbC triple consisting of a pre-/postcondition specification and a statement; a leaf is a concrete statement and intermediate nodes are abstract statements. A refinement is visualized as an edge between two nodes. If the program is fully refined, it can be exported as Java code.

In Fig. 7, we show on the left-hand side an excerpt of the auction example in CorC (cf. Line 14 in Listing 1). We zoomed in to focus on the main features of the editor rather than showing the complete program. The leaf node is selected, which contains the assignment  $i = i + 1$ ; In the properties view, we show the security levels of the variables in the pre- and postcondition and the context. As we can see, we have an assignment to the public variable  $i$ . The calculated least upper bound is private, as we are in a private context, and therefore the security level of  $i$  is updated to private in the postcondition. The outcome of the tool is equal to our calculated security levels in the example above (cf. Fig. 4). In the middle of the graphic, the palette of CorC is shown to add refinements per drag and drop. On the right-hand side, the constructed program in textual form is shown, which is generated automatically by CorC.

This IFbC implementation extends the CorC [25] tool for correctness-by-construction. Besides information flow, CorC reasons about the functional correctness using a functional specification. By refining a program, the pre-/postcondition specification is updated according to the refinement rules and the side conditions are discharged automatically. To separate the functional conditions and the security levels graphically, we decide for a properties view to visualize the information flow at each step in the program. This fits the separation of concerns because the conditions for the functional correctness can be altered by the user, but the information flow is calculated automatically by CorC. By using the refinement rules and analyzing the declared variables, the security level of each variable at each step in the program can be computed. Users do not have to find invariants for loops or other specifications to ensure compliance with the information flow policy. If the user detects an inconsistency in the program, the exact spot where a variable deviates from the intended behavior can be pinpointed.

*Applicability of IFbC.* To demonstrate applicability of IFbC, we have conducted smaller case studies. Users already familiar with CorC were able to create secure programs while ensuring functional correctness simultaneously. As the IFbC rules are applied automatically, users only noticed the security mechanisms whenever they were prevented from writing insecure code.

<sup>1</sup><https://github.com/TUBS-ISF/CorC>

We emphasize that correctness-by-construction is intended to be used in correctness-critical applications [18]. Therefore, the scope of this extension to prevent information leaks is the same. Mostly small security-critical programs will be constructed with IFbC. However, the approach also supports constructing larger programs by splitting them into smaller ones using method calls.

An advantage of IFbC is the constructive nature. Instead of checking that the information flow policy is not violated after writing the program, users can directly construct programs to comply with the policy. In every step of the program, even in partial programs, all variables and their security levels can be observed without executing the program. Another advantage is that the security (confidentiality as well as integrity) and functional correctness are guaranteed simultaneously, as a secure program that does not have the intended behavior is insufficient for the users. Functional correctness is also a mandatory requirement for integrity.

IFbC can be used supplementary to existing standard quality control mechanisms (e.g., a type system, provided that the type system has the same expressiveness, or testing) to increase trust in the created program. A program is constructed with IFbC, and afterwards or at certain points, other mechanisms are used to cross-check the correctness of the program. Overall, the IFbC approach is feasible for creating critical software. As finished programs can be automatically exported to Java, IFbC can be embedded into existing concepts or processes for secure Java development.

The functional CbC tool without information flow was already evaluated qualitatively with a user study [26]. In comparison to a post-hoc verification tool, the participants appreciated the good feedback of CorC to find defects in the code. The additional effort for using the refinement rules, was not mentioned negatively.

## 7 RELATED WORK

In the following, we discuss the differences to prior work and distinguish IFbC from other Hoare-style logics for information flow control. We also discuss information flow type systems and functional correctness-by-construction.

*C14bC.* We build on top of existing work. Schaefer et al. [29] introduced C14bC as a constructive approach to reason about information flow. They introduced a Hoare-style confidentiality specification with two levels, high and low, and developed refinement rules to create programs that ensure this specification. The programs are written in a simple while-language without method calls.

IFbC extends the specification of C14bC from high and low confidentiality levels to a lattice of security levels. We adapted the refinement rules to preserve the security of constructed programs for any input of user defined security levels. Confidentiality, integrity, as well as functional correctness can be ensured simultaneously in one program. We also extended the underlying language with a method call to improve the scalability of the approach, and we proved soundness of all refinement rules. Furthermore, IFbC is implemented as tool support that ensures the lattice-based information flow policy.

*Hoare-Style Logics for Information Flow.* Previous works that use Hoare-style program logics with information flow control analyze the programs after construction, instead of guaranteeing the security during construction. The work of Andrews and Reitman [5] is



## REFERENCES

- [1] Jean-Raymond Abrial. 2010. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press.
- [2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *STTT* 12, 6 (2010), 447–466.
- [3] Torben Amtoft and Anindya Banerjee. 2004. Information Flow Analysis in Logical Form. In *SAS (LNCS)*, Vol. 3148. Springer, 100–115.
- [4] Torben Amtoft, John Hatcliff, Edwin Rodriguez, Robby, Jonathan Hoag, and David A. Greve. 2008. Specification and Checking of Software Contracts for Conditional Information Flow. In *FM*. Springer, 229–245.
- [5] Gregory R. Andrews and Richard P. Reitman. 1980. An Axiomatic Approach to Information Flow in Programs. *TOPLAS* 2, 1 (1980), 56–76.
- [6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traou, Damien Oeteau, and Patrick D. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *PLDI*, Vol. 49. ACM, 259–269.
- [7] Ralph-Johan Back. 2009. Invariant Based Programming: Basic Approach and Teaching Experiences. *FAOC* 21, 3 (2009), 227–244.
- [8] Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. 2007. Testing and Verifying Invariant Based Programs in the SOCOS Environment. In *TAP (LNCS)*, Vol. 4454. Springer, 61–78.
- [9] Ralph-Johan Back and Joakim Wright. 2012. *Refinement Calculus: A Systematic Introduction*. Springer Science & Business Media.
- [10] Anindya Banerjee and David A. Naumann. 2002. Secure Information Flow and Pointer Confinement in a Java-like Language. In *CSFW*, Vol. 2. 253.
- [11] D Elliott Bell and Leonard J La Padula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation*. Technical Report. MITRE Corp Bedford MA.
- [12] Kenneth J Biba. 1977. *Integrity Considerations for Secure Computer Systems*. Technical Report. MITRE Corp Bedford MA.
- [13] Bart De Win, Riccardo Scandariato, Koen Buyens, Johan Grégoire, and Wouter Joosen. 2009. On the Secure Software Development Process: CLASP, SDL and Touchpoints Compared. *InfSof* 51, 7 (2009), 1152–1171.
- [14] Dorothy E Denning. 1976. A Lattice Model of Secure Information Flow. *CACM* 19, 5 (1976), 236–243.
- [15] Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice Hall.
- [16] Michael Howard, Steve Lipner, U Index, U Part, U Chapter, U Why In, U First Steps, U New Threats, U Windows, U Seeking Scalability, et al. 2006. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press.
- [17] Sebastian Hunt and David Sands. 2006. On Flow-Sensitive Security Types. *SIGPLAN Not.* 41, 1 (Jan. 2006), 79–90.
- [18] Derrick G. Kourie and Bruce W. Watson. 2012. *The Correctness-By-Construction Approach to Programming*. Springer.
- [19] Peixuan Li and Danfeng Zhang. 2017. Towards a Flow-and Path-Sensitive Information Flow Analysis. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 53–67.
- [20] Carroll Morgan. 1994. *Programming from Specifications* (2nd ed.). Prentice Hall.
- [21] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *POPL*. ACM, 228–241.
- [22] Andrew C. Myers and Barbara Liskov. 2000. Protecting Privacy Using the Decentralized Label Model. *TOSEM* 9, 4 (2000), 410–442.
- [23] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer.
- [24] Marcel Vinicius Medeiros Oliveira, Ana Cavalcanti, and Jim Woodcock. 2003. ArcAngel: A Tactic Language for Refinement. *FAOC* 15, 1 (2003), 28–47.
- [25] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick Kourie, and Bruce W. Watson. 2019. Tool Support for Correctness-by-Construction. In *FASE (LNCS)*, Vol. 11424. Springer, 25–42.
- [26] Tobias Runge, Thomas Thüm, Loek Cleophas, Ina Schaefer, and Bruce W. Watson. 2019. Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study. In *Refine*. Springer. To appear.
- [27] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, 186–199.
- [28] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *J-SAC* 21, 1 (2003), 5–19.
- [29] Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick Kourie, and Bruce W. Watson. 2018. Towards Confidentiality-by-Construction. In *ISoLA (LNCS)*, Vol. 11244. Springer, 502–515.
- [30] Katja Tuma, Riccardo Scandariato, and Musard Balliu. 2019. Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis. In *JCSA*. IEEE, 191–200.
- [31] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *JCS* 4, 2/3 (1996), 167–188.
- [32] Bruce W. Watson, Derrick G. Kourie, Ina Schaefer, and Loek Cleophas. 2016. Correctness-by-Construction and Post-hoc Verification: A Marriage of Convenience?. In *ISoLA (LNCS)*, Vol. 9952. Springer, 730–748.
- [33] Steve Zdancewic and Andrew C. Myers. 2001. Robust Declassification. In *CSFW*. IEEE, 15–23.

## **A.8. Information Flow Control-by-Construction for an Object-Oriented Language**



# Information Flow Control-by-Construction for an Object-Oriented Language

Tobias Runge<sup>1,2</sup>(✉) , Alexander Kittelmann<sup>1,2</sup> , Marco Servetto<sup>3</sup>,  
Alex Potanin<sup>4</sup> , and Ina Schaefer<sup>1,2</sup>

<sup>1</sup> TU Braunschweig, Braunschweig, Germany

<sup>2</sup> Karlsruhe Institute of Technology, Karlsruhe, Germany

{tobias.runge, alexander.kittelmann, ina.schaefer}@kit.edu

<sup>3</sup> Victoria University of Wellington, Wellington, New Zealand

marco@ecs.vuw.ac.nz

<sup>4</sup> Australian National University, Canberra, Australia

alex.potanin@anu.edu.au

**Abstract.** In security-critical software applications, confidential information must be prevented from leaking to unauthorized sinks. Static analysis techniques are widespread to enforce a secure information flow by checking a program after construction. A drawback of these systems is that incomplete programs during construction cannot be checked properly. The user is not guided to a secure program by most systems. We introduce IFbCOO, an approach that guides users incrementally to a secure implementation by using refinement rules. In each refinement step, confidentiality or integrity (or both) is guaranteed alongside the functional correctness of the program, such that insecure programs are declined by construction. In this work, we formalize IFbCOO and prove soundness of the refinement rules. We implement IFbCOO in the tool CorC and conduct a feasibility study by successfully implementing case studies.

**Keywords:** Correctness-by-construction · Information flow control · Security-by-design

## 1 Introduction

For security-critical software, it is important to ensure *confidentiality* and *integrity* of data, otherwise attackers could gain access to this secure data. For example, in a distributed system, one client A has a lower privilege (i.e., a lower security level) than another client B. When both clients send information to each other, security policies can be violated. If A reads secret data from B, confidentiality is violated. If B reads untrusted data from A, the integrity of B's data is no longer guaranteed. To ensure security in software, mostly static analysis techniques are used, which check the software after development [28]. A violation of

security is only revealed after the program is fully developed. If violations occur, an extensive and repetitive repairing process of writing code and checking the security properties with the analysis technique is needed. An alternative is to check the security with language-based techniques such as type systems [28] during the development. In such a secure type system, every expression is assigned to a type, and a set of typing rules checks that the security policy is not violated [28]. If violations occur, an extensive process of debugging is required until the code is type-checked.

To counter these shortcomings, we propose a constructive approach to directly develop functionally correct programs that are secure by design without the need of a *post-hoc* analysis. Inspired by the correctness-by-construction (CbC) approach for functional correctness [18], we start with a security specification and refine a high-level abstraction of the program stepwise to a concrete implementation using a set of refinement rules. Guided by the security specification defining the allowed security policies on the used data, the programmer is directly informed if a refinement is not applicable because of a prohibited information flow. With IFbCOO (Information Flow control by Construction for an Object-Oriented language), programmers get a local warning as soon as a refinement is not secure, which can reduce debugging effort. With IFbCOO, functionally correct and secure programs can be developed because both, the CbC refinement rules for functional correctness and the proposed refinement rules for information flow security, can be applied simultaneously.

In this paper, we introduce IFbCOO which supports information flow control for an object-oriented language with type modifiers for mutability and alias control [13]. IFbCOO is based on IFbC [25] proposed by some of the authors in previous work, but lifts its programming paradigm from a simple imperative language to an object-oriented language. IFbC introduced a sound set of refinement rules to create imperative programs following an information flow policy, but the language itself is limited to a simple while-language. In contrast, IFbCOO is based on the secure object-oriented language SIFO [27]. SIFO's type system uses immutability and uniqueness properties to facilitate information flow reasoning. In this work, we translate SIFO's typing rules to refinement rules as required by our correctness-by-construction approach. This has the consequence that programs written in SIFO and programs constructed using IFbCOO are interchangeable. In summary, our contributions are the following. We formalize IFbCOO and establish 13 refinement rules. We prove soundness that programs constructed with IFbCOO are secure. Furthermore, we implement IFbCOO in the tool CorC and conduct a feasibility study.

## 2 Object-Oriented Language SIFO by Example

SIFO [27] is an object-oriented language that ensures secure information flow through a type system with precise uniqueness and (im)mutability reasoning. SIFO introduces four type modifiers for references, namely `read`, `mut`, `imm`, and `capsule`, which define allowed aliasing and mutability of objects in programs.

While, `mut` and `imm` point to mutable and immutable object respectively, a `capsule` reference points to a mutable object that cannot be accessed from other `mut` references. A `read` reference points to an object that cannot be aliased or mutated. In this section, SIFO is introduced with examples to give an overview of the expressiveness and the security mechanism of the language. We use in the examples two security levels, namely `low` and `high`. An information flow from `low` to `high` is allowed, whereas the opposite flow is prohibited. The security levels can be arranged in any user-defined lattice. In Sect. 4, we introduce SIFO formally. In Listing 1, we show the implementation of a class `Card` containing a `low` immutable `int` `number` and two `high` fields: a mutable `Balance` and an immutable `Pin`.

```

1 class Card{low imm int number; high mut Balance blc;
2   high imm Pin pin;}
3 class Balance{low imm int blc;}
4 class Pin{low imm int pin;}

```

**Listing 1.** Class declarations

In Listing 2, we show allowed and prohibited field assignments with immutable objects as information flow reasoning is the easiest with these references. In a secure assignment, the assigned expression and the reference need the same security level (Lines 6,7). This applies to mutable and immutable objects. The security level of expressions is calculated by the least upper bound of the accessed field security level and the receiver security level. A `high int` cannot be assigned to a `low blc` reference (Line 8) because this would leak confidential information to an attacker, when the attacker reads the `low blc` reference. The assignment is rejected. Updates of a `high` immutable field are allowed with a `high int` (Line 9) or with a `low int` (Line 10). The `imm` reference guarantees that the assigned integer is not changed, therefore, no new confidential information can be introduced and a promotion in Line 10 is secure. The promotion alters the security level of the assigned expression to be equal to the security level of the reference. As expected, the opposite update of a `low` field with a `high int` is prohibited in Line 11 because of the direct flow from higher to lower security levels.

```

5 low mut Card c = new low Card();//an existing Card reference
6 high mut Balance blc = c.blc;//correct access of high blc
7 high imm int blc = c.blc.blc;//correct access of high blc.blc
8 low imm int blc = c.blc.blc;//wrong high assigned to low
9 c.blc.blc = highInt;//correct field update with high int
10 c.blc.blc = c.number;//correct update with promoted imm int
11 high imm int highInt = 0;//should be some secret value
12 c.number = highInt;//wrong, high int assigned to low c.number

```

**Listing 2.** Examples with immutable objects

Next, in Listing 3, we exemplify which updates of mutable objects are legal and which updates are not. We have a strict separation of mutable objects with



different security levels. We want to prohibit that an update through a higher reference is read by lower references, or that an update through lower references corrupt data of higher references. A new `Balance` object can be initialized as a `low` object because the `Balance` object itself is not confidential (Line 12). The association to a `Card` object makes it a confidential attribute of the `Card` class. However, the assignment of a `low mut` object to a `high` reference is prohibited. If Line 13 would be accepted, Line 14 could be used to insecurely update the confidential `Balance` object because the `low` reference is still in scope of the program. Only an assignment without aliasing is allowed (Line 16). With `capsule`, an encapsulated object is referenced to which no other `mut` reference points. The `low capsule` object can be promoted to a `high` security level and assigned. Afterwards, the `capsule` reference is no longer accessible. In the case of an immutable object, the aliasing is allowed (Line 18), since the object itself cannot be updated (Line 19). Both `imm` and `capsule` references are usable to communicate between different security levels.

```

12 low mut Balance newBlc = new low Balance(0); //ok
13 c.blc = newBlc; //wrong, mutable secret shared as low and high
14 newBlc.blc = 10; //ok? Insecure with previous line
15 low capsule Balance capsBlc = new low Balance(0); //ok
16 c.blc = capsBlc; //ok, no alias introduced
17 low imm Pin immPin = new low Pin(1234); //ok
18 c.pin = immPin; //ok, pin is imm and can be aliased
19 immPin.pin = 5678; //wrong, immutable object cannot be updated

```

**Listing 3.** Examples with mutable and encapsulated objects

### 3 IFbCOO by Example

With IFbCOO, programmers can incrementally develop programs, where the security levels are organized in a lattice structure to guarantee a variety of confidentiality and integrity policies. IFbCOO defines 13 refinement rules to create secure programs. As these rules are based on refinement rules for correctness-by-construction, programmers can simultaneously apply refinements rules for functional correctness [12, 18, 26] and security. We now explain IFbCOO in the following examples. For simplicity, we omit the functional specification. IFbCOO is introduced formally in Sect. 4.

In IFbCOO, the programmer starts with a class including fields of the class and declarations of method headers. IFbCOO is used to implement methods in this class successively. The programmer chooses one abstract method body and refines this body to a concrete implementation of the method. A starting IFbCOO tuple specifies the typing context  $\Gamma$  and the abstract method body  $eA$ . The expression  $eA$  is abstract in the beginning and refined incrementally to a concrete implementation. During the construction process, local variables can be added. The refinement process in IFbCOO results in a method implementation which can be exported to the existing class. First, we give a fine-grained example to show the application of refinement rules in detail. The second example illustrates that IFbCOO can be used to implement larger methods.

The first example in Listing 4 is a setter method. A field `number` is set with a parameter `x`. We start the construction with an abstract expression  $eA : [\Gamma; \text{low imm void}]$  with a typing context  $\Gamma = \text{low mut } C \text{ this}, \text{low imm int } x$  extracted from the method signature ( $C$  is the class of the method receiver). The abstract expression  $eA$  contains all local information (the typing context and its type) to be further refined. A concrete expression that replaces the abstract expression must have the same type `low imm void`, and it can only use variables from the typing context  $\Gamma$ . The tuple  $[\Gamma; \text{low imm void}]$  is now refined stepwise. First, we introduce a field assignment:  $eA \rightarrow eA_1.\text{number} = eA_2$ . The newly introduced abstract expressions are  $eA_1 : [\Gamma; \text{low mut } C]$  and  $eA_2 : [\Gamma; \text{low imm int}]$  according to the field assignment refinement rule. In the next step,  $eA_1$  is refined to `this`, which is the following refinement:  $eA_1.\text{number} = eA_2 \rightarrow \text{this.number} = eA_2$ . As `this` has the same type as  $eA_1$ , the refinement is correct. The last refinement replaces  $eA_2$  with `x`, resulting in  $\text{this.number} = eA_2 \rightarrow \text{this.number} = x$ . As `x` has the same type as  $eA_2$ , the refinement is correct. The method is fully refined since no abstract expression is left.

```

1  low mut method low imm void setNumber(low imm int x) {
2      this.number = x; }

```

Listing 4. Set method

To present a larger example, we construct a check of a signature in an email system (see Listing 5). The input of the method is an `email` object and a `client` object that is the receiver of the email. The method checks whether the key with which the `email` object was signed and the stored public key of the `client` object are a valid pair. If this is the case, the `email` object is marked as verified. The fields `isSignatureVerified` and `emailSignKey` of the class `email` have a high security level, as they contain confidential data. The remaining fields have low as security level.

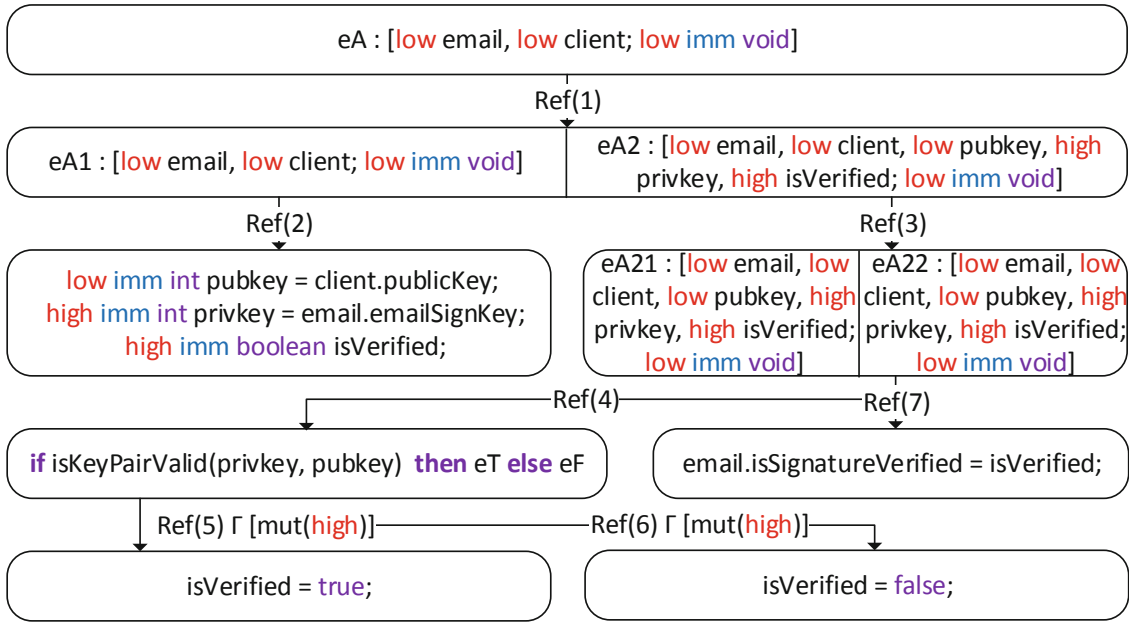
```

1  static low imm void verifySignature(
2      low mut Client client, low mut Email email) {
3      low imm int pubkey = client.publicKey;
4      high imm int privkey = email.emailSignKey;
5      high imm boolean isVerified;
6      if (isKeyPairValid(privkey, pubkey)) {
7          isVerified = true;
8      } else {
9          isVerified = false;
10     }
11     email.IsSignatureVerified = isVerified;
12 }

```

Listing 5. Program of a secure signature verification

In Fig. 1, we show the starting IFbCOO tuple with the security level of the variables (type modifier and class name are omitted) at the top. In our example,



**Fig. 1.** Refinement steps for the signature example

we have two parameters `client` and `email`, with a `low` security level. To construct the algorithm of Listing 5, the method implementation is split into three parts. First, two local variables (private and public key for the signature verification) are initialized and a Boolean for the result of the verification is declared. Second, verification whether the keys used for the signature form a valid pair takes place. Finally, the result is saved in a field of the `email` object.

Using the refinement rule for composition, the program is initially split into the initialization phase and the remainder of the program's behavior (Ref(1)). This refinement introduces two abstract expressions  $eA1$  and  $eA2$ . The typing contexts of the expressions are calculated by IFbCOO automatically during refinement. As we want to initialize two local variables by further refining  $eA1$ , the finished refinement in Fig. 1 already contains the local `high` variables `privkey` and `isVerified`, and the `low` variable `pubkey` in the typing context of expression  $eA2$ .

In Ref(2), we apply the assignment refinement<sup>1</sup> to initialize the integers `pubkey` and `privkey`. Both references point to immutable objects that are accessed via fields of the objects `client` and `email`. The security levels of the field accesses are determined with the field access rule checked by IFbCOO. The determined security level of the assigned expression must match the security level of the reference. In this case, the security levels are the same. Additionally, it is enforced that immutable objects cannot be altered after construction (i.e., it is not possible to corrupt the private and public key). In Ref(3), the next expression  $eA2$  is split with a composition refinement into  $eA21$  and  $eA22$ .

Ref(4) introduces an if-then-else-expression by refining  $eA21$ . Here, it is checked whether the public and private key pair is valid. As the `privkey`

<sup>1</sup> To be precise, it would be a combination of composition and assignment refinements, because an assignment refinement can only introduce one assignment expression.

$$\begin{aligned}
T & ::= s \text{ mdf } C \\
s & ::= \mathbf{high} \mid \mathbf{low} \mid \dots (\text{user defined}) \\
\text{mdf} & ::= \mathbf{mut} \mid \mathbf{imm} \mid \mathbf{capsule} \mid \mathbf{read} \\
CD & ::= \mathbf{class } C \mathbf{ implements } \overline{C} \{ \overline{F} \overline{MD} \} \mid \mathbf{interface } C \mathbf{ extends } \overline{C} \{ \overline{MH} \} \\
F & ::= s \mathbf{ mut } C f; \mid s \mathbf{ imm } C f; \\
MD & ::= MH \{ \mathbf{return } e; \} \\
MH & ::= s \text{ mdf } \mathbf{method } T m(T_1 x_1, \dots, T_n x_n) \\
e & ::= eA \mid x \mid e_0.f = e_1 \mid e.f \mid e_0.m(\overline{e}) \mid \mathbf{new } s C(\overline{e}) \mid e_0; e_1 \\
& \quad \mid \mathbf{if } e_0 \mathbf{ then } e_1 \mathbf{ else } e_2 \mid \mathbf{while } e_0 \mathbf{ do } e_1 \mid \mathbf{declassify}(e) \\
\Gamma & ::= x_1 : T_1 \dots x_n : T_n \\
\mathcal{E} & ::= [] \mid \mathcal{E}.f \mid \mathcal{E}.f = e \mid e.f = \mathcal{E} \mid \mathcal{E}.m(\overline{e}) \mid e.m(\overline{e} \mathcal{E} \overline{e}) \mid \mathbf{new } s C(\overline{e} \mathcal{E} \overline{e})
\end{aligned}$$

**Fig. 2.** Syntax of the extended core calculus of SIFO

object has a **high** security level, we have to restrict our typing context with  $\Gamma[\mathbf{mut}(\mathbf{high})]$ . This is necessary to prevent indirect information leaks. With the restrictions, we can only assign expressions to at least **high** references and mutate **high** objects ( $\mathbf{mut}(\mathbf{high})$ ) in the then- and else-expression. If we assign a value in the then-expression to a **low** reference that is visible outside of the then-expression, an attacker could deduce that the guard was evaluated to true by reading that **low** reference.

Ref(5) introduces an assignment of an immutable object to a **high** reference, which is allowed in the restricted typing context. As explained, the assignment to **low** references is forbidden. The assigned immutable object **true** can be securely promoted to a **high** security level. In Ref(6), a similar assignment is done, but with the value **false**. Ref(7) sets a field of the **email** object by refining  $eA22$ . We update the **high** field of the **email** object by accepting the **high** expression **isVerified**. With this last refinement step, the method is fully concretized. The method is secure by construction and constitutes valid SIFO code (see Listing 5).

## 4 Formalizing Information Flow Control-by-Construction

In this section, we formalize IFbCOO for the construction of functionally correct and secure programs. Before, we introduce SIFO as the underlying programming language formally.

### 4.1 Core Calculus of SIFO

Figure 2 shows the syntax of the extended core calculus of SIFO [27]. SIFO is an expression-based language similar to Featherweight Java [17]. Every reference and expression is associated with a type  $T$ . The type  $T$  is composed of a security level  $s$ , a type modifier  $\text{mdf}$  and a class name  $C$ . Security levels are arranged in a lattice with one greatest level  $\top$  and one least level  $\perp$  forming the security policy. The security policy determines the allowed information flow. Confidentiality and integrity can be enforced by using two security lattices and two security annotations for each expression. Each property is enforced by a

strict separation of security levels. In the interest of an expressive language, we allow the information flow from lower to higher levels (confidentiality or integrity security levels) using promotion rules while the opposite needs direct interaction with the programmer by using the `declassify` expression. For convenience, we will use only one lattice of confidentiality security levels in the explanations.

The type modifier *mdf* can be `mut`, `imm`, `capsule`, and `read` with the following subtyping relation. For all type modifier *mdf* : `capsule`  $\leq$  *mdf*, *mdf*  $\leq$  `read`. In SIFO, objects are *mutable* or (deeply) *immutable*. The reachable object graph (ROG) from a mutable object is composed of mutable and immutable objects, while the ROG of an immutable object can only contain immutable objects. A `mut` reference must point to a mutable object; such an object can be aliased and mutated. An `imm` reference must point to an immutable object; such an object can be aliased, but not mutated. A `capsule` reference points to a mutable object. The object and the mutable objects in its ROG cannot be accessed from other references. As `capsule` is a subtype of `imm` and `mut` the object can be assigned to both. Finally, a `read` reference is the supertype that points to an object that cannot be aliased or mutated, but it has no immutability guarantee that the object is not modified by other references. These modifiers allow us to make precise decisions about the information flow by utilizing immutability or uniqueness properties of objects. For example, an immutable object cannot be altered, therefore it can be securely promoted to a higher security level. For a mutable object, a security promotion is insecure because an update through other references with lower security levels can corrupt the confidential information.

Additionally, the syntax of SIFO contains class definitions *CD* which can be classes or interfaces. An interface has a list of method headers. A class has additional fields. A field *F* has a type *T* and a name, but the type modifier can only be `mut` or `imm`. A method definition *MD* consists of a method header and a body. The header has a receiver, a return type, and a list of parameters. The parameters have a name and a type *T*. The receiver has a type modifier and a security level. An expression *e* can be a variable, field access, field assignment, method call, or object construction in SIFO. In the extended version presented in the paper, we also added abstract expressions, sequence of expressions, conditional expression, loop expression, and declassification. With the `declassify` operator a reverse information flow is allowed. The expression *eA* is abstract and typed by  $[T; T]$ . Beside the type *T* a local typing context *T* is used to have all needed information to further refine *eA*. We require a Boolean type for the guards in the conditional and loop expression. A typing context *T* assigns a type  $T_i$  to variable  $x_i$ . With the evaluation context  $\mathcal{E}$ , we define the order of evaluation for the reduction of the system. The typing rules of SIFO are shown in the report [24].

## 4.2 Refinement Rules for Program Construction

To formalize the IFbCOO refinement rules, in Fig. 3, we introduce basic notations, which are used in the refinement rules.

*L* is the lattice of security levels to define the information flow policy and *lub* is used to calculate the least upper bound of a set of security levels. The

$L$	Bounded upper semi-lattice $(L, \leq)$ of security levels
$\text{lub} : \mathcal{P}(L) \rightarrow L$	Least upper bound of the security levels in $L$
$\{P; Q; \Gamma; T; eA\}$	Starting IFbCOO tuple
$eA : [P; Q; \Gamma; T]$	Typed abstract expression $eA$
$\Gamma[\text{mut}(s)]$	Restricted typing context
$\text{sec}(T) = s$	Returns the security level $s$ in type $T$

**Fig. 3.** Basic notations for IFbCOO

functional and security specification of a program is defined by an IFbCOO tuple  $\{P; Q; \Gamma; T; eA\}$ . The IFbCOO tuple consists of a typing context  $\Gamma$ , a type  $T$ , an abstract expression  $eA$ , and a functional pre-/postcondition, which is declared in the first-order predicates  $P$  and  $Q$ . The abstract expression is typed by  $[P; Q; \Gamma; T]$ . In the following, we focus on security, so the functional specification is omitted.

The refinement process of IFbCOO starts with a method declaration, where the typing context  $\Gamma$  is extracted from the arguments and  $T$  is the method return type. Then, the user guides the construction process by refining the first abstract expression  $eA$ . With the notation  $\Gamma[\text{mut}(s)]$ , we introduce a restriction to the typing context. The function  $\text{mut}(s)$  prevents mutation of mutable objects that have a security level lower than  $s$ . When the user chooses the lowest security level of the lattice, the function does not restrict  $\Gamma$ . The function  $\text{sec}(T)$  extracts the security level of a type  $T$ .

*Refinement Rules.* The refinement rules are used to replace an IFbCOO tuple  $\{\Gamma; T; eA\}$  with a concrete implementation by concretizing the abstract expression  $eA$ . This refinement is only correct if specific side conditions hold. On the right side of the rules, all newly introduced symbols are implicitly existentially quantified. The rules can introduce new abstract expressions  $eA_i$  which can be refined by further applying the refinement rules.

### Refinement Rule 1 (Variable)

$eA$  is refinable to  $x$  if  $eA : [\Gamma; T]$  and  $\Gamma(x) = T$ .

The first IFbCOO rule introduces a variable  $x$ , which does not alter the program. It refines an abstract expression to an  $x$  if  $x$  has the correct type  $T$ .

### Refinement Rule 2 (Field Assignment)

$eA$  is refinable to  $eA_0.f := eA_1$  if  $eA : [\Gamma; T]$  and  $eA_0 : [\Gamma; s_0 \text{ mut } C_0]$  and  $eA_1 : [\Gamma; s_1 \text{ mdf } C]$  and  $s \text{ mdf } C \ f \in \text{fields}(C_0)$  and  $s_1 = \text{lub}(s_0, s)$ .

We can refine an abstract expression to a field assignment if the following conditions hold. The expression  $eA_0$  has to be `mut` to allow a manipulation of the object. The security level of the assigned expression  $eA_1$  has to be equal to the least upper bound of the security levels of expression  $eA_0$  and the field  $f$ . The field  $f$  must be a field of the class  $C_0$  with the type  $s \text{ mdf } C$ . With the security promotion rule, the security level of the assigned expression can be altered.



**Refinement Rule 3 (Field Access)**

$eA$  is refinable to  $eA_0.f$  if  $eA : [\Gamma; s \text{ mdf } C]$  and  $eA_0 : [\Gamma; s_0 \text{ mdf}_0 C_0]$  and  $s_1 \text{ mdf}_1 C$   $f \in \text{fields}(C_0)$  and  $s = \text{lub}(s_0, s_1)$  and  $\text{mdf}_0 \triangleright \text{mdf}_1 = \text{mdf}$ .

We can refine an abstract expression to a field access if a field  $f$  exists in the class of receiver  $eA_0$  with the type  $s_1 \text{ mdf}_1 C$ . The accessed value must have the expected type  $s \text{ mdf } C$  of the abstract expression. This means, the class name of the field  $f$  and  $C$  must be the same. Additionally, the security level of the abstract expression  $eA$  is equal to the least upper bound of the security levels of expression  $eA_0$  and field  $f$ . The type modifiers must also comply. The arrow between type modifiers is defined as follows. As we allow only `mut` and `imm` fields, not all possible cases are defined:  $\text{mdf} \triangleright \text{mdf}' = \text{mdf}''$

- $\text{mut} \triangleright \text{mdf} = \text{capsule} \triangleright \text{mdf} = \text{mdf}$
- $\text{imm} \triangleright \text{mdf} = \text{mdf} \triangleright \text{imm} = \text{imm}$
- $\text{read} \triangleright \text{mut} = \text{read}$ .

**Refinement Rule 4 (Method Call)**

$eA$  is refinable to  $eA_0.m(eA_1, \dots, eA_n)$  if  $eA : [\Gamma; T]$  and  $eA_0 : [\Gamma; T_0] \dots eA_n : [\Gamma; T_n]$  and  $T_0 \dots T_n \rightarrow T \in \text{methTypes}(\text{class}(T_0), m)$  and  $\text{sec}(T) \geq \text{sec}(T_0)$  and for all  $i \in \{1, \dots, n\}$  if  $\text{mdf}(T_i) \in \{\text{mut}, \text{capsule}\}$  then  $\text{sec}(T_i) \geq \text{sec}(T_0)$ .

With the method call rule, an abstract expression is refined to a call to method  $m$ . The method has a receiver  $eA_0$ , a list of parameters  $eA_1 \dots eA_n$ , and a return value. A method with matching definition must exist in the class of receiver  $eA_0$ . This method definition is returned by the `methTypes` function. The function `class` returns the class of a type  $T$ . The security level of the return type has to be greater than or equal to the security level of the receiver. This condition is needed because through dynamic dispatch information of the receiver may be leaked if its security level is higher than the security level of the return type. The same applies for `mut` and `capsule` parameters. The security level of these parameters must also be greater than or equal to the security level of the receiver. As the method call replaces an abstract expression  $eA$ , the return value must have the same type (security level, type modifier, and class name) as the refined expression. In the technical report [24], we introduce multiple methods types [27] to reduce writing effort and increase the flexibility of IFbCOO. A method can be declared with specific types for receiver, parameters and return value, and other signatures of this method are deduced by applying the transformations from the multiple method types definition, where security level and type modifiers are altered. All these deduced method declarations can be used in the method call refinement rule.

**Refinement Rule 5 (Constructor)**

$eA$  is refinable to `new`  $s C(eA_1 \dots eA_n)$  if  $eA : [\Gamma; s \text{ mdf } C]$  and  $\text{fields}(C) = T_1 f_1 \dots T_n f_n$  and  $eA_1 : [\Gamma; T_1[s]] \dots eA_n : [\Gamma; T_n[s]]$ .

The constructor rule is a special method call. We can refine an abstract expression to a constructor call, where a mutable object of class  $C$  is constructed



with a security level  $s$ . The parameter list  $eA_1 \dots eA_n$  must match the list of declared fields  $f_1 \dots f_n$  in class  $C$ . Each parameter  $eA_i$  is assigned to field  $f_i$ . This assignment is allowed if the type of parameter  $eA_i$  is (a subtype of)  $T_i[s]$ .  $T[s]$  is a helper function which returns a new type whose security level is the least upper bound of  $sec(T)$  and  $s$ . It is defined as:  $T[s] = lub(s, s') \text{ mdf } C$ , where  $T = s' \text{ mdf } C$ , defined only if  $s' \leq s$  or  $s \leq s'$ . By calling a constructor, the security level  $s$  can be freely chosen to use parameters with security levels that are higher than originally declared for the fields. In other words, a security level  $s$  is used to initialize lower security fields with parameters of higher security level  $s$ . This results in a newly created object with the security level  $s$  [27]. As the newly created object replaces an abstract expression  $eA$ , the object must have the same type as the abstract expression. If the modifier promotion rule is used (i.e., no mutable input value exist), the object can be assigned to a **capsule** or **imm** reference.

**Refinement Rule 6 (Composition)**

$eA$  is refinable to  $eA_0$ ;  $eA_1$  if  $eA : [\Gamma; T]$  and  $eA_0 : [\Gamma; T_0]$  and  $eA_1 : [\Gamma; T]$ .

With the composition rule, an abstract expression  $eA$  is refined to two subsequent abstract expression  $eA_0$  and  $eA_1$ . The second abstract expression must have the same type  $T$  as the refined expression.

**Refinement Rule 7 (Selection)**

$eA$  is refinable to **if**  $eA_0$  **then**  $eA_1$  **else**  $eA_2$  if  $eA : [\Gamma; T]$  and  $eA_0 : [\Gamma; s \text{ imm Boolean}]$  and  $eA_1 : [\Gamma[mut(s)]; T]$  and  $eA_2 : [\Gamma[mut(s)]; T]$ .

The selection rule refines an abstract expression to a conditional **if-then-else**-expression. Secure information can be leaked indirectly as the selected branch may reveal the value of the guard. In the branches, the typing context is restricted. The restricted typing context prevents updating mutable objects with a security level lower than  $s$ . The security level  $s$  is determined by the Boolean guard  $eA_0$ . When we add updatable local variables to our language, the selection rule must also prevent the update of local variables that have a security level lower than  $s$ .

**Refinement Rule 8 (Repetition)**

$eA$  is refinable to **while**  $eA_0$  **do**  $eA_1$  if  $eA : [\Gamma; T]$  and  $eA_0 : [\Gamma; s \text{ imm Boolean}]$  and  $eA_1 : [\Gamma[mut(s)]; T]$ .

The repetition rule refines an abstract expression to a **while**-loop. The repetition rule is similar to the selection rule. For the loop body, the typing context is restricted to prevent indirect leaks of the guard in the loop body. The security level  $s$  is determined by the Boolean guard  $eA_0$ .

**Refinement Rule 9 (Context Rule)**

$\mathcal{E}[eA]$  is refinable to  $\mathcal{E}[e]$  if  $eA$  is refinable to  $e$ .

The context rule replaces in a context  $\mathcal{E}$  an abstract expression with a concrete expression, if the abstract expression is refinable to the concrete expression.

**Refinement Rule 10 (Subsumption Rule)**

$eA : [\Gamma; T]$  is refinable to  $eA_1 : [\Gamma; T']$  if  $T' \leq T$ .

The subsumption rule can alter the type of expressions. An abstract expression that requires a type  $T$  can be weakened to require a type  $T'$  if the type  $T'$  is a subtype of  $T$ .

**Refinement Rule 11 (Security Promotion)**

$eA : [\Gamma; s \text{ mdf } C]$  is refinable to  $eA_1 : [\Gamma; s' \text{ mdf } C]$  if  $\text{mdf} \in \{\text{capsule}, \text{imm}\}$  and  $s' \leq s$ .

The security promotion rule can alter the security level of expressions. An abstract expression that requires a security level  $s$  can be weakened to require a security level  $s'$  if the expression is `capsule` or `imm`. Other expressions (`mut` or `read`) cannot be altered because potentially existing aliases are a security hazard.

**Refinement Rule 12 (Modifier Promotion)**

$eA : [\Gamma; s \text{ capsule } C]$  is refinable to  $eA_1 : [\Gamma[\text{mut} \setminus \text{read}]; s \text{ mut } C]$ .

The modifier promotion rule can alter the type modifier of an expression  $eA$ . An abstract expression that requires a `capsule` type modifier can be weakened to require a `mut` type modifier if all `mut` references are only seen as `read` in the typing context. That means, that the mutable objects in the ROG of the expression cannot be accessed by other references. Thus, manipulation of the object is only possible through the reference on  $eA$ .

**Refinement Rule 13 (Declassification)**

$eA : [\Gamma; \perp \text{ mdf } C]$  is refinable to  $\text{declassify}(eA_1) : [\Gamma; s \text{ mdf } C]$  if  $\text{mdf} \in \{\text{capsule}, \text{imm}\}$ .

In our information flow policy, we can never assign an expression with a higher security level to a variable with a lower security level. To allow this assignment in appropriate cases, the `declassify` rule is used. An expression  $eA$  is altered to a `declassify`-expression with an abstract expression  $eA_1$  that has a security level  $s$  if the type modifier is `capsule` or `imm`. A `mut` or `read` expression cannot be declassified as existing aliases are a security hazard. Since we have the security promotion rule, the declassified `capsule` or `imm` expression can directly be promoted to any higher security level. Therefore, it is sufficient to use the bottom security level in this rule without restricting the expressiveness. For example, the rule can be used to assign a hashed password to a public variable. The programmer has the responsibility to ensure that the use of `declassify` is secure.

### 4.3 Proof of Soundness

In the technical report, we prove that programs constructed with the IFbCOO refinement rules are secure according to the defined information flow policy. We

prove this by showing that programs constructed with IFbCOO are well typed in SIFO (Theorem 1). SIFO itself is proven to be secure [27]. In the technical report [24], we prove this property for the core language of SIFO, which does not contain composition, selection, and repetition expressions. The SIFO core language is minimal, but using well-known encodings, it can support composition, selection, and repetition (encodings of the Smalltalk [14] style support control structures). We also exclude the declassify operation because this rule is an explicit mechanism to break security in a controlled way.

**Theorem 1 (Soundness of IFbCOO)**

*An expression  $e$  constructed with IFbCOO is well typed in SIFO.*

## 5 CorC Tool Support and Evaluation

IFbCOO is implemented in the tool CorC [12,26]. CorC itself is a hybrid textual and graphical editor to develop programs with correctness-by-construction. IFbC [25] is already implemented as extension of CorC, but to support object-orientation with IFbCOO a redesign was necessary. Source code and case studies are available at: <https://github.com/TUBS-ISF/CorC/tree/CCorCOO>.

### 5.1 CorC for IFbCOO

For space reasons, we cannot introduce CorC comprehensively. We just summarize the features of CorC to check IFbCOO information flow policies:

- Programs are written in a tree structure of refining IFbCOO tuples (see Fig. 1). Besides the functional specification, variables are labeled with a type  $T$  in the tuples.
- Each IFbCOO refinement rule is implemented in CorC. Consequently, functional correctness and security can be constructed simultaneously.
- The information flow checks according to the refinement rules are executed automatically after each refinement.
- Each CorC-program is uniquely mapped to a method in a SIFO class. A SIFO class contains methods and fields that are annotated with security labels and type modifiers.
- A properties view shows the type  $T$  of each used variable in an IFbCOO tuple. Violations of the information flow policy are explained in the view.

### 5.2 Case Studies and Discussion

The implementation of IFbCOO in the tool CorC enables us to evaluate the feasibility of the security mechanism by successfully implementing three case studies [16,32] from the literature and a novel one in CorC. The case studies are also implemented and type-checked in SIFO to confirm that the case studies are secure. The newly developed *Database* case study represents a secure system

**Table 1.** Metrics of the case studies

Name	#Security levels	# Classes	# Lines of code	# Methods in CorC
Database	4	6	156	2
Email [16]	2	9	807	15
Banking [32]	2	3	243	6
Paycard	2	3	244	5

that strictly separates databases of different security levels. *Email* [16] ensures that encrypted emails cannot be decrypted by recipients without the matching key. *Paycard* (<http://spl2go.cs.ovgu.de/projects/57>) and *Banking* [32] simulate secure money transfer without leaking customer data. The *Database* case study uses four security levels, while the others (*Email*, *Banking*, and *Paycard*) use two.

As shown in Table 1, the cases studies comprise three to nine classes with 156 to 807 lines of code each. 28 Methods that exceed the complexity of getter and setter are implemented in CorC. It should be noted that we do not have to implement every method in CorC. If only `low` input data is used to compute `low` output, the method is intrinsically secure. For example, three classes in the Database case study are implemented with only `low` security levels. Only the class `GUI` and the main method of the case study, which calls the `low` methods with higher security levels (using multiple method types) is then correctly implemented in CorC. The correct and secure promotion of security levels of methods called in the main method is confirmed by CorC.

*Discussion and Applicability of IFbCOO.* We emphasize that CbC and also IFbCOO should be used to implement correctness- and security-critical programs [18]. The scope of this work is to demonstrate the feasibility of the incremental construction of correctness- and security-critical programs. We argue that we achieve this goal by implementing four case studies in CorC.

The constructive nature of IFbCOO is an advantage in the secure creation of programs. Instead of writing complete methods to allow a static analyzer to accept/reject the method, with IFbCOO, we directly design and construct secure methods. We get feedback during each refinement step, and we can observe the status of all accessible variables at any time of the method. For example, we received direct feedback when we manipulated a `low` object in the body of a `high` then-branch. With this information, we could adjust the code to ensure security. As IFbCOO extends CorC, functional correctness is also guaranteed at the same time. This is beneficial as a program, which is security-critical, should also be functionally correct. As IFbCOO is based on SIFO, programs written with any of the two approaches can be used interchangeably. This allows developers to use their preferred environment to develop new systems, re-engineer their systems, or integrate secure software into existing systems. These benefits of IFbCOO are of course connected with functional and security specification effort, and the strict refinement-based construction of programs.

## 6 Related Work

In this section, we compare IFbCOO to IFbC [25, 29] and other Hoare-style logics for information flow control. We also discuss information flow type systems and correctness-by-construction [18] for functional correctness.

IFbCOO extends IFbC [25] by introducing object-orientation and type modifiers. IFbC is based on a simple while language. As explained in Sect. 4, the language of IFbCOO includes objects and type modifiers. Therefore, the refinement rules of IFbC are revised to handle secure information flow with objects. The object-orientation complicates the reasoning of secure assignments because objects could be altered through references with different security levels. If private information is introduced, an already public reference could read this information. SIFO and therefore IFbCOO consider these cases and prevent information leaks by considering immutability and encapsulation and only allowing secure aliases.

Previous work using Hoare-style program logics with information flow control analyzes programs after construction, rather than guaranteeing security during construction. Andrews and Reitman [5] encode information flow directly in a logical form. They also support parallel programs. Amtoft and Banerjee [3] use Hoare-style program logics and abstract interpretations to detect information flow leaks. They can give error explanations based on strongest postcondition calculation. The work of Amtoft and Banerjee [3] is used in SPARK Ada [4] to specify and check the information flow.

Type system for information flow control are widely used, we refer to Sabelfeld and Myers [28] for a comprehensive overview. We only discuss closely related type systems for object-oriented languages [9–11, 20, 30, 31]. Banerjee et al. [9] introduced a type system for a Java-like language with only two security levels. We extend this by operating on any lattice of security levels. We also introduce type modifiers to simplify reasoning in cases where objects cannot be mutated or are encapsulated. Jif [20] is a type system to check information flow in Java. One main difference is in the treatment of aliases: Jif does not have an alias analysis to reason about limited side effects. Therefore, Jif pessimistically discards programs that introduce aliases because Jif has no option to state immutable or encapsulated objects. IFbCOO allows the introduction of secure aliases.

In the area of correctness-by-construction, Morgan [19] and Back [8] propose refinement-based approaches which refine functional specifications to concrete implementations. Beside of pre-/postcondition specification, Back also uses invariants as starting point. Morgan’s calculus is implemented in ArcAngel [22] with the verifier ProofPower [33], and SOCOS [6, 7] implements Back’s approach. In comparison to IFbCOO, those approaches do not reason about information flow security. Other refinement-based approaches are Event-B [1, 2] for automata-based systems and Circus [21, 23] for state-rich reactive systems. These approaches have a higher abstraction level, as they operate on abstract machines instead of source code. Hall and Chapman [15] introduced with CbyC another related approach that uses formal modeling techniques to analyze the develop-

ment during all stages (architectural design, detailed design, code) to eliminate defects early. IFbCOO is tailored to source code and does not consider other development phases.

## 7 Conclusion

In this paper, we present IFbCOO, which establishes an incremental refinement-based approach for functionally correct and secure programs. With IFbCOO programs are constructed stepwise to comply at all time with the security policy. The local check of each refinement can reduce debugging effort, since the user is not warned only after the implementation of a whole method. We formalized IFbCOO by introducing 13 refinement rules and proved soundness by showing that constructed programs are well-typed in SIFO. We also implemented IFbCOO in CorC and evaluated our implementation with a feasibility study. One future direction is the conduction of comprehensive user studies for user-friendly improvements which is only now possible due to our sophisticated tool CorC.

**Acknowledgments.** This work was supported by KASTEL Security Research Labs.

## References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transfer* **12**(6), 447–466 (2010)
3. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 100–115. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-27864-1\\_10](https://doi.org/10.1007/978-3-540-27864-1_10)
4. Amtoft, T., Hatcliff, J., Rodríguez, E.: Specification and checking of software contracts for conditional information flow. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 229–245. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-68237-0\\_17](https://doi.org/10.1007/978-3-540-68237-0_17)
5. Andrews, G.R., Reitman, R.P.: An axiomatic approach to information flow in programs. *ACM Trans. Program. Langu. Syst. (TOPLAS)* **2**(1), 56–76 (1980)
6. Back, R.J.: Invariant based programming: basic approach and teaching experiences. *Formal Aspects Comput.* **21**(3), 227–244 (2009)
7. Back, R.-J., Eriksson, J., Myreen, M.: Testing and verifying invariant based programs in the SOCOS environment. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 61–78. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-73770-4\\_4](https://doi.org/10.1007/978-3-540-73770-4_4)
8. Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (2012)
9. Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a Java-like language. In: Computer Security Foundations Workshop, vol. 2, p. 253 (2002)




10. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference Java bytecode verifier. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-71316-6\\_10](https://doi.org/10.1007/978-3-540-71316-6_10)
11. Barthe, G., Serpette, B.P.: Partial evaluation and non-interference for object calculi. In: Middeldorp, A., Sato, T. (eds.) FLOPS 1999. LNCS, vol. 1722, pp. 53–67. Springer, Heidelberg (1999). [https://doi.org/10.1007/10705424\\_4](https://doi.org/10.1007/10705424_4)
12. Bordis, T., Cleophas, L., Kittelmann, A., Runge, T., Schaefer, I., Watson, B.W.: Re-CorC-ing KeY: correct-by-construction software development based on KeY. In: Ahrendt, W., Beckert, B., Bubel, R., Johnsen, E.B. (eds.) The Logic of Software. A Tasting Menu of Formal Methods. LNCS, vol. 13360, pp. 80–104. Springer, Cham (2022). [https://doi.org/10.1007/978-3-031-08166-8\\_5](https://doi.org/10.1007/978-3-031-08166-8_5)
13. Giannini, P., Servetto, M., Zucca, E., Cone, J.: Flexible recovery of uniqueness and immutability. *Theor. Comput. Sci.* **764**, 145–172 (2019)
14. Goldberg, A., Robson, D.: *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc. (1983)
15. Hall, A., Chapman, R.: Correctness by construction: developing a commercial secure system. *IEEE Softw.* **19**(1), 18–25 (2002)
16. Hall, R.J.: Fundamental nonmodularity in electronic mail. *Autom. Softw. Eng.* **12**(1), 41–79 (2005)
17. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **23**(3), 396–450 (2001)
18. Kourie, D.G., Watson, B.W.: *The Correctness-by-Construction Approach to Programming*. Springer, Heidelberg (2012)
19. Morgan, C.: *Programming from Specifications*, 2nd edn. Prentice Hall, Hoboken (1994)
20. Myers, A.C.: JFlow: practical mostly-static information flow control. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 228–241. ACM (1999)
21. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for circus. *Formal Aspects Comput.* **21**(1), 3–32 (2009)
22. Oliveira, M.V.M., Cavalcanti, A., Woodcock, J.: ArcAngel: a tactic language for refinement. *Formal Aspects Comput.* **15**(1), 28–47 (2003)
23. Oliveira, M.V.M., Gurgel, A.C., Castro, C.G.: CRefine: support for the circus refinement calculus. In: *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pp. 281–290. IEEE (2008)
24. Runge, T., Kittelmann, A., Servetto, M., Potanin, A., Schaefer, I.: Information flow control-by-construction for an object-oriented language using type modifiers (2022). <https://arxiv.org/abs/2208.02672>
25. Runge, T., Knüppel, A., Thüm, T., Schaefer, I.: Lattice-based information flow control-by-construction for security-by-design. In: *Proceedings of the 8th International Conference on Formal Methods in Software Engineering* (2020)
26. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool support for correctness-by-construction. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 25–42. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2)
27. Runge, T., Servetto, M., Potanin, A., Schaefer, I.: Immutability and Encapsulation for Sound OO Information Flow Control (2022, under review)
28. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE J. Sel. Areas Commun.* **21**(1), 5–19 (2003)



29. Schaefer, I., Runge, T., Knüppel, A., Cleophas, L., Kourie, D., Watson, B.W.: Towards confidentiality-by-construction. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 502–515. Springer, Cham (2018). [https://doi.org/10.1007/978-3-030-03418-4\\_30](https://doi.org/10.1007/978-3-030-03418-4_30)
30. Strecker, M.: Formal analysis of an information flow type system for MicroJava. Technische Universität München, Technical report (2003)
31. Sun, Q., Banerjee, A., Naumann, D.A.: Modular and constraint-based information flow inference for an object-oriented language. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 84–99. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-27864-1\\_9](https://doi.org/10.1007/978-3-540-27864-1_9)
32. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-based deductive verification of software product lines. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering, pp. 11–20 (2012)
33. Zeyda, F., Oliveira, M., Cavalcanti, A.: Supporting ArcAngel in ProofPower. *Electron. Notes Theor. Comput. Sci.* **259**, 225–243 (2009)





Technische Universität Carolo-Wilhelmina Braunschweig  
Institut für Softwaretechnik und Fahrzeuginformatik

Mühlenpfordtstr. 23  
D-38106 Braunschweig