Correctness-by-Construction: An Overview of the CorC Ecosystem

Tabea Bordis
tabea.bordis@kit.edu
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

Tobias Runge
tobias.runge@kit.edu
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

Alexander Kittelmann
alexander.kittelmann@kit.edu
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

Ina Schaefer
ina.schaefer@kit.edu
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

Abstract

Correctness-by-Construction (CbC) is an incremental software development technique in the field of formal methods to create functionally correct programs guided by a specification. In contrast to post-hoc verification, where the specification and verification take part after implementing a program, with CbC the specification is defined first, and then the program is successively created using a small set of refinement rules that define side conditions preserving the correctness of the program. This specification-first, refinement-based approach as pursued by CbC has the advantage that errors are likely to be detected earlier in the design process and can be tracked more easily. Even though the idea of CbC emerged over 40 years ago, CbC is not widespread and is mostly used to create small algorithms. We believe in the idea of CbC and envision a scaled CbC approach that contributes to solving problems of modern software verification. In this short paper, we give an overview of our research regarding CbC in four different lines of research. For all of them, we provide tool support for building the CorC ecosystem that even further enables CbC-based development for different fields of application and size of software systems. Furthermore, we give an outlook on future work that extends on our concepts for CbC.

Keywords: correctness-by-construction, information flow control, software product lines, architecture, program verification

1 Introduction

The amount of software in safety-critical systems increases, and, therefore, functional correctness of programs is an important concern. While most verification approaches rely on post-hoc verification, where a program is only verified *after* it is implemented, the incremental approach of *Correctness-by-Construction* (CbC) as imagined by Dijkstra [10], Gries [11], or Kourie and Watson [14] offers an alternative.[1] CbC starts with defining a formal specification in an abstract Hoare triple `{P} S {Q}` consisting of a precondition `P`, an abstract statement `S`, and a postcondition `Q`. Hoare triples represent total correctness assertions that are only `true` if, starting from the precondition, the postcondition is met after executing the eventually defined concrete program. With CbC, a Hoare triple is successively refined using a set of refinement rules to a concrete implementation, which satisfies the specification. To guarantee the correctness of the refinement steps, each rule defines specific side conditions for its applicability.

The underlying idea of this specification-first, refinement-based approach is that better programs can be constructed when the developer must think about their construction more thoroughly rather than hacking them into correctness. As a result, when applying CbC compared to classical post-hoc verification, errors are more likely to be detected earlier in the design process [14]. Additionally, programmers and users gain trust because a formal methodology was used to create the program.

To further investigate these claims and spread the idea of correct-by-construction software development, we implemented CbC in the tool CorC [18]. CorC is a graphical and textual IDE to construct algorithms following CbC. It supports developers to refine a program by a sequence of

[1] CbC as we pursue it is different from correctness-by-construction (CbyC) as promoted by Hall and Chapman [12]. CbyC is a software development process where formal modeling techniques are used to make it difficult to introduce defects and to detect and remove any defects that do occur as early as possible.

refinement steps and to verify the correctness of these refinement steps using the theorem prover KeY [2]. First evaluation results show a decreased verification effort compared to post-hoc verification [6, 18].

Besides CbC as we pursue it, there are also other tools that implement different refinement-based approaches. For example, Event-B [1] uses automata-based system descriptions instead of source code as done by CorC, ArcAngel [15] is based on Morgan's refinement calculus and comprises a very large number of refinement rules compared to minimal set in CorC, and SOCOS [3] uses invariants as specifications instead of pre- and postcondition pairs as used in CorC.

In this short paper, we give an overview of the CorC ecosystem[2] that combines existing research on improving and extending the applicability of CbC to emphasize the benefit of the ecosystem as a whole. We present four lines of research where CbC is either considered for different fields of application or integrated into software engineering processes to scale its applicability from single algorithms to object-oriented software systems and component-based architectures. All of these lines of research are implemented as extensions of CorC constituting the CorC ecosystem to benefit the community. Last, we give an outlook on our vision for future research on CbC for quantitative information flow control by construction.

2 The CorC Ecosystem

In the past years, we have been investigating different fields of application for CbC that benefit from the idea of a structured development process guided by specifications and refinements. First, we investigated how CbC that has been designed to create single, small algorithms can be used in modern software engineering processes. We therefore integrated object-oriented programming as a commonly used paradigm into CorC and improved the development process of CorC to enable the development using CbC other verification techniques in concert. A natural follow-up is to develop concepts and tool support that make CbC available at scale. Currently, we aim to address this in two further directions. First, we study the role of correct-by-construction implementations in software architectures with ArchiCorC, where the main goal is to bundle CorC programs into reusable software components. Second, VarCorC is a framework for CbC-based development of software product lines. Instead of developing monolithic programs, the goal of software product lines is to systematically construct a family of similar software programs following the CbC paradigm. Last, we applied CbC-style refinement rules to ensure security of programs. Therefore, we introduced an information flow policy to CbC (IFbC) and implemented it in an extension of CorC. We briefly present all these lines of research in the following sections.

[2]The CorC ecosystem is available at https://github.com/TUBS-ISF/CorC

Object-Oriented Development using Correctness-by-Construction

The size and complexity of software rapidly increases. Therefore, software engineering paradigms need to adapt to these requirements. Guaranteeing correctness for these complex systems is still a challenge. To scale the applicability of CbC, we extended CorC to support object-oriented programming and investigated a software engineering process in CorC to use CbC in concert with other verification strategies or classical testing [5]. Object-oriented programming introduces classes with fields and class invariants to CbC, which allows to develop more complex projects including inheritance and interfaces. At the same time, CbC may not be the ideal method to guarantee correctness for a large software project. Therefore, we support a roundtrip engineering process from Java code to CbC development to correct Java code. We integrated these concepts and further usability-features that simplify the development using CbC in the successor of CorC, CorC 2.0.[3]

Correct-by-Construction Software Architectures

Component-based architectures allow to establish a set of reusable, correct-by-construction components. This is equally interesting for libraries, where implementations are accessed through interfaces, and for third-party developments that are easier to integrate into individual projects. Most important, creating components that modularize correct implementations allows developers to think about how to compose software systems instead of how to program a monolithic software system from scratch. We argue that this is the foundation for building large and complex systems that are based on CbC. As an extension to CorC, we propose a framework called ArchiCorC [13] that connects UML-style component modeling, formal specification, and code generation. ArchiCorC[4] comprises the following key ingredients. First, a component and interface description language is used to interconnect provided and required interfaces of components. Second, developers can either refine method signatures of provided interfaces to correct implementations using CorC , or map signatures to already existing CorC programs. Third, analyses and algorithms to check compatibility between components are provided. Finally, ArchiCorC allows to generate Java code from the correct-by-construction components.

Correctness-by-Construction for Software Product Lines

Software product lines provide systematic reuse paired with variability mechanisms to realize whole product families [9]. The commonalities and differences of the product variants

[3]https://github.com/TUBS-ISF/CorC/tree/CorC2.0, CorC 2.0 supports object-oriented development and development for software product lines using CbC (VarCorC)

[4]https://github.com/TUBS-ISF/ArchiCorC

are communicated as features, whose relationships are often modeled in feature models. Guaranteeing the correctness of a product line is especially challenging because of the number of possible product variants resulting from the number of feature configurations and the variable code structures [20]. To create a correct product line using CbC, we extended the original CbC approach with a new refinement rule for a variability mechanism that allows to call different implementations of a method depending on the distinct feature configuration [6, 7]. Additionally, we combined this mechanism with contract composition for variability in the pre- and postcondition [8]. We call this extension *variational Correctness-by-Construction*. VarCorC[2] uses FeatureIDE [21] and variational CbC to support the development of correct-by-construction software product lines.

Information Flow Control-by-Construction

Besides verifying functional correctness, it is also important to consider non-functional properties of a program, such as dependability, reliability, resource/energy consumption, or security [4]. For security, an information flow policy can be used to define how information may flow in a program (e.g., a flow from public to secret data is allowed, but the other way is prohibited to ensure confidentiality and integrity of the data). Our extension of CbC to ensure this type of security-by-design is called *Information Flow Control-by-Construction* (IFbC) [17]. Programs are constructed incrementally using refinement rules to follow an information flow policy. In every refinement step, security and functional correctness of the program is guaranteed, such that insecure programs are prohibited by construction. The information flow policy can be specified in any bounded upper semi-lattice (i.e., security levels are arranged in a lattice representing the allowed direction of information flow). IFbC is implemented in an extension of CorC.[5]

Implementation of the CorC Ecosystem

The core of the CorC ecosystem is the tool CorC [18] which is an open-source Eclipse plug-in supporting the development of programs with CbC. It stores the structure of a CbC program including the refinement rules through a meta-model that is modeled using the Eclipse Modeling Framework.[6] CorC comes with a graphical and textual editor. The graphical editor is implemented using Graphiti[7] and visualizes the underlying meta-model in a tree structure. The textual editor is implemented using XText.[8] The beginning of a CbC program is a Hoare triple, which can then be refined by applying CbC refinement rules until there are no more abstract statements. In the background, the deductive verification tool KeY [2] is used to prove the correct application of each refinement rule.

The presented lines of research each extend the core functionality of CorC. For example, VarCorC uses FeatureIDE [21] to integrate feature models and calculations on them [7], while CorC 2.0 introduces another graphical view for classes with fields and implements a roundtrip engineering process that generates correct Java code from CbC programs [5]. Further implementation details are provided in the referenced papers and on GitHub.

3 Correctness-by-Construction - Next Steps

Driven by our research in the past years that we conducted on fields of application and extensions of CorC as tool, we can see our vision of scaling CbC as necessary practice in modern software engineering come together. We are convinced that CbC in combination with good tool support is an underestimated approach that is worth exploring. This belief is also substantiated by the participants of two user studies that agree that CorC is good tool to develop correct software [16, 19]. Besides the extensions that we presented in the previous section, there are still some open ideas that we want to explore in future work on CbC.

Feature Interactions in CbC Product Lines.

Naturally, features in a software product line interact with each other using shared variables or by calling methods defined in other features. While most of these feature interactions are wanted or even needed to implement functionality, sometimes there are also unwanted feature interactions that lead to malfunctions, unexpected behavior, or security leaks. An example of an unwanted feature interaction in an Email product line is when a feature that automatically forwards incoming mails to a certain address and a feature that decrypts incoming mails are combined together in a software product. In that case, an email may first be decrypted and afterwards forwarded in plain text to the forward receiver which violates a security property of encrypting mails. In future work, we will examine unwanted feature interactions and develop concepts to integrate safety and security constraints to the functional specification used in VarCorC, our extension of CbC for software product lines. Our goal is to give a guarantee that there are no unwanted feature interactions in a product line constructed with CbC.

Quantitative Information Flow Control

Confidentiality and integrity requirements on data, as well as privacy concerns can be expressed using information flow control policies, specifying how data may flow through a program and which observations an attacker may make about the data that is being processed. For future work, we will work on our vision of CbC to enable security-by-design with

[5] https://github.com/TUBS-ISF/CorC/tree/CCorC
[6] https://eclipse.org/emf/
[7] https://eclipse.org/graphiti/
[8] https://www.eclipse.org/Xtext/

an extension of IFbC for quantitaive information flow. We will extend IFbC with quantitative and probabilistic information flow specifications, and develop refinement rules for correct program construction using appropriate program annotations and side conditions. We will further investigate how probabilistic programming constructs can be used to capture uncertainty of program execution and what influence they have on the information flow specifications, both classically and quantitative. To enable scalability of the correctness-by-construction engineering process, we will extend existing work on the correct construction of component-based systems with classical and quantitative information flow policies to obtain larger functionally correct systems incorporating security-by-design.

References

[1] Jean-Raymond Abrial. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. 2016. *Deductive Software Verification – The KeY Book*. Springer.

[3] Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. 2007. Testing and Verifying Invariant Based Programs in the SOCOS Environment. In *International Conference on Tests and Proofs*. Springer.

[4] Maurice H ter Beek, Loek Cleophas, Ina Schaefer, and Bruce W Watson. 2018. X-by-Construction. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 359–364.

[5] Tabea Bordis, Loek Cleophas, Alexander Kittelmann, Tobias Runge, Ina Schaefer, and Bruce W Watson. 2022. Re-CorC-ing KeY: Correct-by-Construction Software Development Based on KeY. In *The Logic of Software. A Tasting Menu of Formal Methods*. Springer, 80–104.

[6] Tabea Bordis, Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. 2020. Variational Correctness-by-Construction. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. 1–9.

[7] Tabea Bordis, Tobias Runge, and Ina Schaefer. 2020. Correctness-by-Construction for Feature-Oriented Software Product Lines. In *International Conference on Generative Programming: Concepts and Experiences*. 22–34.

[8] Tabea Bordis, Tobias Runge, David Schultz, and Ina Schaefer. 2022. Family-based and Product-based Development of Correct-by-Construction Software Product Lines. *Journal of Computer Languages* (2022), 101119.

[9] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Citeseer.

[10] Edsger W. Dijkstra. 1976. *A Discipline of Programming* (1st ed.). Prentice Hall PTR.

[11] David Gries. 1981. *The Science of Programming* (1st ed.). Springer.

[12] Anthony Hall and Roderick Chapman. 2002. Correctness by Construction: Developing a Commercial Secure System. *IEEE software* 19, 1 (2002), 18–25.

[13] Alexander Knüppel, Tobias Runge, and Ina Schaefer. 2020. Scaling Correctness-by-Construction. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 187–207.

[14] Derrick G. Kourie and Bruce W. Watson. 2012. *The Correctness-by-Construction Approach to Programming*. Springer.

[15] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. 2003. ArcAngel: A Tactic Language for Refinement. *Formal Aspects of Computing* (2003).

[16] Tobias Runge, Tabea Bordis, Thomas Thüm, and Ina Schaefer. 2021. Teaching Correctness-by-Construction and Post-hoc Verification–The Online Experience. In *Formal Methods Teaching Workshop*. Springer, 101–116.

[17] Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. 2020. Lattice-based Information Flow Control-by-Construction for Security-by-Design. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*. 44–54.

[18] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick Kourie, and Bruce W. Watson. 2019. Tool Support for Correctness-by-Construction. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 25–42.

[19] 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.

[20] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* (2014).

[21] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79, 0 (2014), 70–85.