Karlsruhe Institute of Technology

Test, Validation, Analysis

Scaling Correctness-by-Construction

Zur Erlangung des akademischen Grades einer

Doktorin der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Tabea Bordis

aus Braunschweig

Tag der mündlichen Prüfung: 17. Juli 2025
1. Referentin: Prof. Dr.-Ing. Ina Schaefer
2. Referent: Prof. Dr. Einar Broch Johnsen

Abstract

As software in safety-critical systems, such as automotive, aviation, and medical systems, grows in complexity, stronger correctness guarantees beyond traditional testing are essential, as faults may endanger human lives. Typically, the functional correctness of such systems is ensured with *post-hoc verification*, which means that a program is verified *after* it has been completely implemented and specified with a pre- and postcondition contract. Usually, automated program verifiers are used for post-hoc verification. A major drawback of post-hoc verification is the lack of construction guidance, making fault localization difficult when verification fails. Faults may result from incorrect implementations, insufficient specifications, or unclosable proofs, leading to trial-and-error adjustments in the specification and code. Ideally, with a guided approach to construct correct software, the developer would think about the problem first rather than a concrete implementation and is then guided towards a correct solution without trial-and-error adjustments.

Correctness-by-Construction (CbC) integrates specification and verification into an incremental development process. Starting from a formal specification as an abstract Hoare triple, programs are progressively refined into verified implementations, checking the correctness of each refinement step. CbC provides benefits, such as early error detection and increased trust in the software through the structured development process. CbC is implemented in the tool CorC and first experiments with CorC show an increased proof efficiency compared to post-hoc verification for small algorithms. Despite these benefits, CbC has still mostly been applied to isolated algorithms. However, in modern software engineering, complex, large-scale software systems are developed in teams with different expertise. In particular, we identified four key challenges for applying CbC in modern software engineering: 1) structural complexity through object-orientation, 2) integration into existing development processes, 3) a high entry barrier due to limited verification expertise of developers, and 4) lack of support for advanced programming paradigms, such as those for highly configurable software systems.

In this thesis, we made three contributions that address these four challenges to extend CbC such that it is applicable in modern software engineering. First, we scale CbC to support *object-oriented systems* by introducing an object model with classes, fields, methods, inheritance, class invariants, and frames enabling encapsulation, inheritance, polymorphism, and dynamic dispatch. A roundtrip engineering process enables integration into established software development processes, such that CbC development can be combined with other quality assurance techniques. Second, to lower the entry barrier of using CbC for non-experts in software verification, we integrate a three-level process of increasing correctness guarantees that incorporates *testing* as an intermediate level between *specifying* and *verifying*. Since testing is a well-known quality assurance technique, developers get understandable test reports and can locate and fix major errors early in the design phase. We evaluate the three-level process quantitatively to assess its efficiency in locating faults and qualitatively to asses usability improvements. Third, we extend CbC to support *highly configurable software systems*. The challenge of highly configurable software systems is that they incorporate variability to enable the efficient development of whole product families. Therefore, we incorporate variability in the specification, the code constructs, and the verification strategies of CbC. We prove the soundness of these variability concepts and evaluate feasibility and proof efficiency of the verification strategies. Furthermore, we want to assess whether there is a trade-off between specification effort and proof efficiency when comparing CbC and post-hoc verification. Overall, our work enables CbC to be applied for object-oriented software and highly configurable software systems within software teams with varying expertise levels that use established software development processes.

Zusammenfassung

In sicherheitskritischen Domänen wie dem Automobil-, Flug- und Medizinsektor haben Software-Systeme hohe Sicherheitsanforderungen, da Softwarefehler in solchen Systemen Menschenleben gefährden können. Standardmäßig kommen hier Techniken aus dem Bereich der formalen Methoden, insbesondere die *Post-hoc-Verifikation*, zum Einsatz. Dabei wird ein Programm zunächst vollständig implementiert und formal spezifiziert und erst anschließend auf seine funktionale Korrektheit verifiziert. Ein großer Nachteil von Post-hoc-Verifikation ist, dass Fehler nur schwer lokalisiert werden können, da es an einer strukturierten Vorgehensweise zur Konstruktion der Programme mangelt. Das führt dazu, dass der Code und die Spezifikation oft willkürlich verändert werden, bis sich das Programm schließlich (möglicherweise) verifizieren lässt. Idealerweise kommt es aber gar nicht erst zu einem solchen Trial-and-Error-Vorgehen, wenn ein Entwicklungsansatz verwendet wird, der das Spezifizieren und Verifizieren direkt in den Entwicklungsprozess integriert und somit zu einer korrekten Lösung hinleitet.

Correctness-by-Construction (CbC) integriert Spezifikation und Verifikation in einen inkrementellen Entwicklungsprozess. Ausgehend von einer formalen Spezifikation in Form einer Vor- und Nachbedingung wird das Programm schrittweise zu einer als korrekt beweisbaren Implementierung verfeinert. Dabei wird jeder Verfeinerungsschritt auf seine Korrektheit überprüft. Die Vorteile von CbC sind, dass durch den inkrementellen Prozess Fehler früher entdeckt werden und dass Entwickler:innen dem entwickelten Programm mehr vertrauen, da es in einem formalen Prozess erstellt wurde. Erste Experimente mit dem Tool CorC, welches CbC unterstützt, zeigen eine erhöhte Verifikationseffizienz im Vergleich zu Post-hoc-Verifikation. Dennoch wurde CbC bislang nur zur Entwicklung einzelner Algorithmen angewendet. Moderne Software ist jedoch größer und komplexer und wird in Teams mit verschiedenen Expertisen entwickelt. Das Ziel dieser Dissertation ist es daher, CbC für die Entwicklung im modernen Software-Engineering anwendbar zu machen. Insbesondere fokussiert sich diese Dissertation auf vier konkrete Herausforderungen: 1) Strukturelle Komplexität durch Objekt-Orientierung, 2) Integration in etablierte Softwareentwicklungsprozesse, 3) eine hohe Einstiegshürde durch fehlende Verifikationsexpertise und 4) fehlende Unterstützung von fortgeschrittenen Programmierparadigmen, wie zum Beispiel für hochkonfigurierbarer Software.

In dieser Dissertation werden drei Kernbeiträge vorgestellt, die die zuvor genannten Herausforderungen adressieren. Im ersten Kernbeitrag wird CbC um ein *Objektmodell* erweitert, welches Klassen, Felder, Methoden, Vererbung, Klasseninvarianten und

Methodenframes integriert. Darüber hinaus wird ein *Roundtrip Engineering Prozess* entwickelt, der die Integration von CbC in etablierte Softwareentwicklungsprozesse sowie die Kombination mit weiteren Qualitätssicherungsmethoden ermöglicht. Im zweiten Kernbeitrag wird die Einstiegshürde für Entwickler:innen mit geringen Vorkenntnissen im Bereich der Softwareverifikation verringert. Dazu wird *Testen* als Zwischenstufe zwischen *Spezifizieren* und *Verifizieren* in einem dreistufigen Entwicklungsprozess integriert. Testen ist eine weitverbreitete Qualitätssicherungsmethode, die leicht verständliche Testberichte liefert und somit eine schnelle Fehlersuche und -behebung ermöglicht. Der zweite Kernbeitrag wird hinsichtlich seiner Effektivität, Fehler zu lokalisieren, sowie hinsichtlich Verbesserungsmöglichkeiten der Nutzbarkeit unseres Tools evaluiert. Der dritte Kernbeitrag erweitert CbC um die Entwicklung von *hochkonfigurierbarer Software*, welche Variabilitätsmechanismen nutzt, um eine komplette Softwarefamilie effizient zu implementieren. Daher wird CbC um Variabilitätskonstrukte in der Spezifikation, den Programmkonstrukten und den Verifizierungsstrategien erweitert. Die vorgestellten Konzepte werden auf ihre Korrektheit hin bewiesen. Des Weiteren wird eine Machbarkeitsevaluation durchgeführt und die Beweiseffizienz der vorgestellten Verifizierungsstrategien ermittelt. Darüber hinaus wird überprüft, welcher Zusammenhang zwischen dem Spezifikationsaufwand und der Beweiseffizienz besteht, indem CbC mit Post-hoc-Verifikation verglichen werden. Zusammenfassend ermöglichen die vorgestellten Kernbeiträge eine Anwendung von CbC in objekt-orientierten und hochkonfigurierbaren Softwaresystemen, die mit etablierten Softwareentwicklungsprozessen und von Entwickler:innen mit variierender Expertise im Bereich Softwareverifikation entwickelt werden.

Acknowledgments

Many people supported me in different ways throughout my PhD. With the following words, I would like to express my gratitude for the support, guidance, time, friendship, fun, and love I received.

First of all, I would like to thank Ina Schaefer for giving me the opportunity to pursue my PhD in her research group. She was an outstanding mentor, both personally and academically, and always encouraged me whenever I struggled to believe in myself. I am also grateful for the inspiring working environment she created that allowed me to grow while still leaving room for fun and jokes. Furthermore, I am grateful to Prof. Einar Broch Johnsen for agreeing to be the second reviewer of my doctoral thesis and the audit committee members Prof. Barbara Bruno, Prof. Henning Meyerhenke, and Prof. Michael Beigl. Thank you for your time, effort, and feedback. I am also grateful to Rustan Leino for giving me the opportunity to do an internship at AWS in Seattle and collaborating on a paper with me afterwards.

I am especially grateful for the amazing colleagues I had the privilege to work with at both the group for *Software Engineering and Automotive Informatics* at TU Braunschweig and the group for *Test, Validation and Analysis of Software-Intensive Systems* at Karlsruhe Institute of Technology. I will never forget the wonderful times we shared at Christmas parties, work trips, karaoke nights, pub quizzes, table soccer matches, and tea and coffee breaks. A huge thank you as well for preparing the best and most personal defense party for me. From my colleagues, I want to especially thank the formal methods subgroup members Tobias Runge, Alexander Kittelmann, Maximilian Kodetzki, Asmae Heydari Tabar, Rasmus Rønneberg, and Fynn Demmler with whom I worked together the closest, and Lukas Birkemeyer and Philip Ochs who are my favorite colleagues for coffee breaks apart from the formal methods subgroup. I am truly lucky that some colleagues have even become close friends, with whom I shared not only work but all of life's highs and lows, from challenges and lessons learned to travels and joyful moments.

Finally, I want to thank my family and friends for their tremendous support. I especially want to thank my parents, Verena and Lutz, and my brother Noah for their lifelong love and support. Thank you for a carefree childhood, for supporting me in my education, and for always being there for me. Thank you for everything! I also want to thank my friends from Wolfenbüttel, Braunschweig, and Karlsruhe for checking on me during stressful times and providing a much-needed balance. I want to thank Jules, Laura, Lauro, and Anika for being my biggest cheerleaders and for literally dancing on table tops with me, Lea for being the most understanding long-term friend, always

knowing when to listen, and advising me in all my decisions, Burger for regularly checking on me and sending me emotional support pictures of Urmel, Anna for asking the right questions and drinking espresso martinis with me, Wolfram and Michael for encouraging me to regularly go bouldering, my former flatmates, Pepe, Sai, Max, and Franzi, for all the fun evenings filled with food and games, and making me feel at home when I first moved to Karlsruhe, and Wojciech for exploring Seattle and surroundings with me during our internship. Last but not least, I want to thank my amazing boyfriend Marc who went through all the highs and lows of my PhD with me. Thank you for making me laugh every single day, cooking for me when I had no time, and even moving to a different city with me. I love you!

All of the above contributed to my doctoral thesis and my PhD journey, shaping me as a person and giving me countless unforgettable memories. With the words of Taylor Swift: *I was enchanted to meet you* – and have you all in my life!

Contents

List of Figures

List of Tables

Listings

Part I.

Introduction and Preliminaries

1. Introduction

The amount of software in safety-critical systems is increasing, making functional correctness an important concern. Especially in sectors such as automotive, aviation, and medical, software faults can lead to significant financial losses and even endanger human lives. In aviation, regulations already require correctness guarantees using techniques stronger than software testing for high design assurance levels (RTCA DO-178C, 2011). In this thesis, we focus on deductive verification as technique to guarantee functional correctness for programs specified with formal contracts.

Deductive verification is usually applied *post-hoc*, which means that the specification and implementation are completed first, and afterwards the program is verified with regard to the specification (Meyer, 1992). For the specification, design-by-contract is state-of-the-art to specify programs at method level (Meyer, 1988; Meyer, 1992). Methods are specified with a pre- and postcondition contract. This means that if and only if the method is called in a state that fulfills the precondition, the method terminates, and then the program is in a state described by the postcondition. To implement a provably correct program, a developer[1] either starts by defining a specification or writing code, but there is no guideline on how to construct the program. When a method has been implemented and specified, the code is verified against the specification. For this step, an automated program verifier, such as KeY (Ahrendt et al., 2016a), or a verification-aware programming language, such as Dafny (Leino, 2010), can be used.

The problem with a verification process without a guideline on how to construct a program is that once the verification fails, it is difficult to locate the fault in the program. Non-conformance between code and specifications can have three reasons: 1) an incorrect method implementation, 2) an incorrect or insufficient formal specification, or 3) the prover was unable to (semi-)automatically prove conformance. Most times, this leads to a behavior where arbitrary changes are applied to code and specification, and automatic verification is triggered multiple times until finally the automatic program verifier might succeed (Runge et al., 2020). We call this behavior *iterative trial-and-error adjustments to achieve correctness*, in contrast to the desired behavior where the developer thinks about the problem first and is then guided to the correct solution. Knüppel et al. (2018) report that expertise also plays an important role for verifying programs beyond toy examples, since the difficulty of proving correctness increases with the complexity of the implementation.

[1] In this thesis, we assume that one developer specifies, implements, and verifies the program, but these tasks could also be done by separate persons.

Correctness-by-Construction (CbC) as imagined by Dijkstra (1976), Gries (1981), or Morgan (1990)[2] offers an alternative to design-by-contract and post-hoc verification where the correct program is directly constructed with the specification in an incremental process. 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`. With CbC, a Hoare triple is successively refined using a set of refinement rules to a concrete implementation that satisfies the specification. To guarantee the correctness of the refinement steps, each rule defines specific side conditions that preserve correctness during refinement. The program construction is finished when no abstract statement is left.

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 applying iterative trial-and-error changes to achieve correctness, as we have described above. As a result, when applying CbC compared to state-of-the-art post-hoc verification, errors are more likely to be detected earlier in the design process (Kourie and Watson, 2012; Watson et al., 2016). Previously, CbC was implemented in the tool CorC to assess the benefits of a CbC development style (Runge et al., 2019). In two user studies comparing CbC in the tool CorC with post-hoc verification in the tool KeY (Ahrendt et al., 2016a), the participants highlighted that they generally liked the development style of CbC (Runge et al., 2020; Runge et al., 2021), but with both tools, CorC and KeY, they still struggled to prove some tasks.

Despite these aforementioned benefits of CbC, it has still mostly been applied to single algorithms. Modern software engineering, however, considers large-scale and complex software systems where the correctness of the system is one main concern. Furthermore, software systems are developed in teams where the developers have different expertise. Examining how formal method techniques and tools can be improved to meet the requirements of modern software engineering for both technical and qualitative aspects is an important field of research with a promising impact on practical applications. This thesis aims to investigate how CbC can be scaled to larger systems and collaborative software development settings, addressing both technical and qualitative challenges.

[2] CbC as we pursue it is different from Correctness by Construction (CbyC) as promoted by Hall and Chapman (2002). 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.

1.1. Challenges for Applying Correctness-by-Construction in Modern Software Systems

As CbC has not yet been established as an alternative to post-hoc verification, we discuss the challenges of applying CbC to the development of modern software systems. First ideas about CbC have already been proposed early in 1976 (Dijkstra, 1976), where software was still less complex. The idea was to develop single algorithms in a way that is more thoughtful and accompanied by specifications such that the correctness can be guaranteed. Since 1976, the amount, size, and complexity of software systems have increased. Consequently, software is now developed in teams with diverse areas of expertise, using different quality assurance techniques to address different correctness concerns across system components. CbC has not been adapted for modern software engineering, as described above. In the following, we describe four challenges that we identified for CbC as implemented in CorC to be an alternative to post-hoc verification.

Challenge 1: CbC misses scalability in form of a programming model with objects so that it can be applied to larger systems than single algorithms.

One of the core programming paradigms is *object-orientation*, which is part of many popular programming languages, such as C++, C#, and Java. Usually, classes are used to define fields and methods, and an object is created by instantiating the class with concrete values for the fields. An object then has access to its own fields and methods. Fields and methods can also be declared as being accessible from the outside. Object-orientation is a basic paradigm to model software such that it is modularized and, therefore, easier to maintain and extend. Moreover, it can also be used to model the architecture of a software system, which scales the size of software systems that can be developed even further.

To make CbC applicable in modern software engineering, it is mandatory that classes with fields and methods can be constructed following CbC. Therefore, it is not enough that classes can be defined, but we also need specifications, such as class invariants and method framing (a set of fields that is allowed to be modified by a method), to be able to verify properties of the methods inside the class, but also method calls from outside.

Challenge 2: Tool support for CbC misses an integrated software development process such that it can be used in concert with other tools.

While scalability addresses the structural complexity of modern software (Challenge 1), seamless integration into existing software development processes is equally important to be established as an alternative quality assurance technique. Nowadays, software is contained in many fields, such as automotive, aviation, medical, or banking. All of these systems can be complex and big depending on the use case. Especially safety-critical systems, such as cars or airplanes, can have higher requirements on the quality assurance. But usually not all components of a software system have the same safety level. For example, in a car, the functionality of a brake has a higher safety level than the infotainment system. Therefore, standards, such as the ISO 26262-1:2018 (2018) for functional safety of road vehicles, assign different safety levels to the different parts of the system, which means that one part of a system might have to be verified and another only tested. Furthermore, different quality assurance techniques have different advantages and disadvantages. It is therefore essential that CbC as implemented in CorC can be applied with other quality assurance techniques and tools, such as testing, in concert. This is especially important because the refinement-based development style of CbC and the graphical user interface of CorC are different from programming in one of the common IDEs.

Challenge 3: CbC misses a low entry barrier, such that also non-experts in software verification can use CbC.

In industry, testing is still state-of-art when it comes to quality assurance. According to a GitHub page where people list companies that apply formal methods with links to papers or project websites[3], only around 60 companies worldwide apply formal methods to verify their projects. Examples for this are large IT companies, such as Amazon[4], Google (Erbsen et al., 2020), and Microsoft (Chaudhuri et al., 2010), or companies in aerospace, such as NASA or the Boeing company. Most other companies do not verify their software projects, even if they develop safety-critical systems. We argue that one factor might be that the usability gap between testing tools and formal verification tools is a key barrier, as the latter often require expert knowledge and provide less intuitive feedback. This hypothesis is supported by two user studies that showed that non-experts in software verification, i.e., computer science students with little background in software verification, had difficulties verifying simple algorithms with both, CorC and KeY (Runge et al., 2020; Runge et al., 2021). Hence, we identify the challenge that CbC needs a lower entry barrier to be used in modern software engineering.

Challenge 4: CbC misses support for advanced software engineering techniques that are used in industry, such as highly configurable software.

Recently, the demand for custom-tailored software increased, which led to a software paradigm called *software product line engineering*. Software product lines provide systematic software reuse to implement whole product families efficiently (Pohl et al., 2005). Their application domain includes safety-critical systems where the demand for behavioral correctness is generally higher to reduce the risk of dangerous situations in-field. However, the high variability of software product lines is still a big challenge for formal methods in general. We argue that the stepwise development approach of CbC is beneficial compared to post-hoc verification approaches in dealing with the

[3] https://github.com/ligurio/practical-fm?tab=readme-ov-file, accessed 18.02.2025

[4] https://aws.amazon.com/de/blogs/opensource/lean-into-verified-software-development/, accessed 18.02.2025

growing variability. However, CbC does not contain any notion of variability, neither in the refinement rules nor in the specifications to construct correct software product lines on the code level.

State-of-the-Art

While these challenges outline practical obstacles for applying CbC, several *refinement-based approaches and tools* have been proposed in the past. In the following, we position CbC and CorC as tool within the field of refinement-based software construction approaches, which are based on a refinement calculus and apply correctness-preserving refinement steps.

The Event-B framework (Abrial, 2010) refines automata-based system descriptions to concrete implementations. This approach is implemented in the Rodin platform (Abrial et al., 2010). Unlike Event-B, which operates on system-level automata, CbC focuses on code-level specifications, making the abstraction level and application area different. The language ArcAngel (Oliveira et al., 2003) is based on Morgan's refinement calculus (Morgan, 1990) and implemented with ProofPower (Zeyda et al., 2009). Morgan's refinement calculus comprises a broad set of refinement rules in comparison to the minimal set of refinement rules in CorC. The refinement rules are bundled in tactics that are applied to refine a starting specification to a concrete implementation, discharging proof obligations for every refinement that have to be proven to preserve correctness. Back et al. (2007) developed the tool SOCOS to teach refinement-based programming. In comparison to CbC in CorC, SOCOS works with invariants in contrast to pre-/postcondition specifications as used in CbC and CorC. In user studies, Back (2009) found that a refinement-based development style can be overpowering for developers if tool support is not usable. Furthermore, defining loop invariants has been the biggest issue, as the participants used most of the time to improve an incorrect or partial loop invariant. Dafny (Leino, 2010) is a verification-aware programming language that combines specification and implementation into an automated verification workflow. The developer is free in which order specifications and implementations are written and how fine-granular specifications are designed. It is possible to derive a program from a starting specification with small refinement steps, as prescribed with CbC, however, this is not specifically enforced (Ettinger, 2021).

None of the previously mentioned refinement-based software construction methods and tools have been adopted as an alternative to post-hoc verification. In the case of Dafny, we assume that it lacks specialized tool support for a refinement-based development style. For the other mentioned methods and tools, we argue that the underlying refinement calculi may be overly complex compared to the refinement calculus proposed by Kourie and Watson (2012), which was specifically designed with a focus on comprehensibility and usability. Moreover, the applicability of refinement-based construction approaches in modern software engineering may be hindered by the absence of advanced development paradigms, such as software product line engineering, a challenge that we have also identified for CbC as implemented in CorC. Finally, a non-negligible factor is usability and integrability in existing software development processes, which have not been sufficiently examined and addressed.

While refinement-based development ensures correctness through (manual) stepwise transformation of a starting specification, a related field that emerged from the same ideas of Gries (1981) and Dijkstra (1976) is *program synthesis* where the task is to *automatically* generate a correct program from a starting specification. Foundational work has been proposed by Manna and Waldinger (1980) and has been continued by many others. For example, Stickel et al. (1994) propose a deductive approach that extracts a `Fortran` program from a user-given graphical specification by composing entries of subroutine libraries. Gulwani et al. (2010) propose a component-based synthesis that generates and resolves so-called synthesis constraints and apply their approach to bitvector programs. Heisel (1992) uses a proof system that builds up a proof during program development. Polikarpova et al. (2016) propose an approach to synthesize recursive programs from a specification in the form of a polymorphic refinement type. In contrast to program synthesis, CbC as we apply it does not automatically synthesize code from a specification. CbC is rather a development approach that is guided by a specification and guarantees correctness by proving that the side conditions that are introduced by the set of refinement rules are fulfilled. The developer therefore still has control over the program, while with program synthesis, one of possibly many implementations that fulfill the specification is generated. Furthermore, program synthesis has limitations regarding scalability, as for example recursive programs including loops are hard to synthesize.

1.2. Research Questions

Motivated by the challenges that we identified in the previous section, we formulate our main research question as follows:

Main Research Question

How can we scale Correctness-by-Construction from developing isolated algorithms to be applicable in modern software engineering?

To narrow down this main research question, we divide it into three sub-research questions that we explain in the following.

RQ1: How can we scale Correctness-by-Construction to object-oriented programs, and how can we incorporate Correctness-by-Construction into an integrated software development process?

Even though CbC is supposed to lead to code with a low defect rate and improve the traceability of errors, it is not widespread. As mentioned before, CbC has been proposed when computer science was still a relatively young field. Nowadays, in modern software engineering, many programming languages are based on object-orientation to modularize the pieces of software into manageable units. This has the advantage, amongst many more, that software systems can be scaled and modeled in larger structures. Adapting CbC such that it can be used in object-oriented systems is the base to enable the application of CbC to many more advanced concepts, such as distributed systems, configurable systems, and model-based software development. Another challenge that we observe in larger software systems is that they have parts that are more safety-critical than others. Consequently, quality assurance techniques and tools need to work in concert with other quality assurance techniques and tools. With this research question **RQ1**, we want to address Challenges 1 and 2, and introduce an object model for CbC and integrate CbC into a development process for combined use with other quality assurance tools. Both, the object model and the integrated development process are realized in terms of tool support in CorC.

RQ2: How can we lower the entry barrier for non-experts in by-Construction development?

A refinement-based development style, as enforced with CbC, is quite different from standard programming as it is more guided. Furthermore, testing is state-of-the-art and, therefore, in general more known than formal methods. As a result, expert knowledge is still needed to use CbC and the tool CorC. At the same time, experts in formal methods are rare, especially in industry, because formal methods is a field that is seldomly taught. With this research question **RQ2**, we want to address Challenge 3 and investigate whether integrating practices from standard software engineering into the development process of CbC can decrease the entry barrier. By integrating automated test case generation from formal specifications, we provide a familiar quality assurance method for non-experts, bridging the gap between testing and formal verification. Our goal is to broaden the user group of CbC such that software developers in industry can apply CbC as implemented in CorC. We want to evaluate this quantitatively and qualitatively.

RQ3: How can we extend Correctness-by-Construction to support the development of highly configurable software systems?

Software product line engineering is increasingly used to address the growing demand for custom-tailored software products. Software product lines are modeled in terms of end-user visible features that are implemented efficiently in one code base, from which single products can be generated. However, software product lines are also used to realize safety-critical systems where guaranteeing the correctness of all products is a problem because the number of products is growing exponentially with the number of features. With this research question **RQ3**, we want to address Challenge 4, and extend CbC such that it can be used to specify, implement, and verify variable code structures that are part of software product line realization techniques. Thereby, the stepwise refinement-based development style might be better suited in dealing with the growing variability than post-hoc verification as variability has to be thought of during software construction. We want to evaluate this research question by providing soundness proofs for the new refinement rules and verification approaches, and a quantitative comparison to post-hoc verification.

1.3. Main Contributions

The contributions of this thesis follow the three dimensions of scaling CbC that we identified in the research questions: scalability to larger systems (Bordis et al., 2022a; Bordis et al., 2024), accessibility for non-experts (Bordis et al., 2025 (Submitted)), and support for highly configurable software systems (Bordis et al., 2020a; Bordis et al., 2020b; Bordis et al., 2022b; Bordis et al., 2023a). Apart from the conceptual contributions, we implement all contributions in the tool CorC and its extensions.

Contribution 1: CorC 2.0: Objects and an Integrated Development Process.

Addressing RQ1, we present CorC 2.0., which extends CorC's programming model with objects as used in object-oriented programming. We propose to extend CorC by five features that allow the creation of object-oriented Java programs using CbC. First, we implement a graphical editor to create classes with fields, class invariants, and methods. Second, we support the constructive development of interface and inheritance using the Liskov principle. Third, we include a framing condition for methods. Fourth, we implement a roundtrip engineering process such that existing Java classes can be imported into CorC where their correctness can be shown, and exported back into the original Java project as verified Java code. Thereby, the developer can freely decide which parts are constructed using CbC and which parts are implemented and tested or verified with other methods (e.g., depending on the safety criticality of the part). Fifth, we use a change tracking mechanism that simplifies the development process by marking refinement steps that have to be re-verified due to changes in other contracts or method implementations. The feasibility of this concept is evaluated by means of three subject systems. Furthermore, we compare the proof steps and verification time between CbC as implemented in CorC 2.0 and post-hoc verification as implemented in KeY.

Contribution 2: Three Increasing Levels of Correctness Guarantees, with Test Case Generation as Intermediate Level between Specifying and Verifying.

To lower the entry barrier for non-experts in software verification and to address RQ2, we propose a three-level process of increasing correctness guarantees – *specified*, *tested*, *verified* – to introduce user-friendly feedback early on in the development process. At the level *specified*, the intended behavior of the program is formally defined, supported with error messages and debugging tips. At the level *tested*, test cases are automatically generated from a formal specification of the previous level to detect coarse-grained errors with concrete input and output values. At the level *verified*, the side conditions of the refinement rules are verified. If the verification at this level fails, a counter example is generated to support the developer in analyzing the open proof goals. We evaluate the three-level process in terms of effectiveness with a mutation-based analysis and in terms of usability improvements with a qualitative usability refinement study.

Contribution 3: Correct-by-Construction Software Product Lines.

Addressing RQ3, we propose *variational CbC* which has been extended in three directions. First, we propose a feature-based specification technique matching the fine-grained specifications of CbC and variable predicates that are bound to the features of a product line. Second, we propose additional refinement rules to implement variability in code. Third, we propose a product-based and a family-based verification strategy of showing the correctness of all product variants in the product line. Fourth, we present our tool VarCorC as an extension of CorC implementing our approach to develop CbC product lines. We give proofs of soundness for the new refinement rules and the verification strategies. Furthermore, we evaluate the feasibility of our approach on three subject systems and compare the verification time and needed proof steps to post-hoc verification with KeY.

1.4. Structure of this Thesis

We first give background on contract-based verification in the form of design-by-contract and post-hoc verification, the Correctness-by-Construction approach, and software product lines in Chapter 2. We also introduce a running example that we use throughout this thesis to motivate and explain most concepts.

The main part of this thesis is ordered by its contributions:

CorC 2.0: Objects and an Integrated Development Process. In Chapter 3, we present five key concepts to realize object-orientation in CbC and present CorC 2.0 implementing these key features. This also includes a roundtrip engineering process to make CorC usable in concert with other quality assurance tools.

Three Increasing Levels of Correctness Guarantees. In Chapter 4, we explain the three-level correctness guarantee process for CbC where we introduce test case generation as an intermediate level *tested* between the already existing correctness levels *specified* and *verified*. Further, we present how we support each correctness level in CorC.

Correct-by-Construction Software Product Lines. In Chapter 5, we propose variational CbC which consists of an extension of the specification, the refinement rules, and two verification strategies for the verification of all product variants in a product line. Furthermore, we present the extension of CorC, VarCorC, which implements all of these concepts.

Finally, we conclude this thesis and present future work in Chapter 6.

1.5. New and Previously Published Contributions

Some contributions in this thesis have been published and peer-reviewed. Some other contributions are currently under review, and other contributions have been written solely for this thesis. This section is dedicated to attribution.

Chapter 3 is based on (Bordis et al., 2022a). Chapter 4 is based on (Bordis et al., 2025 (Submitted)). Chapter 5 is largely based on four publications (Bordis et al., 2020a; Bordis et al., 2020b; Bordis et al., 2022b; Bordis et al., 2023a) with the addition of variable predicates as specification technique that has not been published. A list of all publications this thesis is based on, and the related publications, is given in the following.

Publications This Thesis is Based On

Bordis, Tabea, Loek Cleophas, Alexander Kittelmann, Tobias Runge, Ina Schaefer, and Bruce 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. Cham: Springer International Publishing, pp. 80–104. DOI: 10.1007/978-3-031-08166-8_5.

Bordis, Tabea, Maximilian Kodetzki, Tobias Runge, and Ina Schaefer (2023a). “VarCorC: Developing Object-Oriented Software Product Lines Using Correctness-by-Construction”. In: *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops*. Vol. 13765. Cham: Springer International Publishing, pp. 156–163. DOI: 10.1007/978-3-031-26236-4_13.

Bordis, Tabea, Maximilian Kodetzki, and Ina Schaefer (July 2024). “From Concept to Reality: Leveraging Correctness-by-Construction for Better Algorithm Design”. In: *Computer* 57.7, pp. 113–119. DOI: 10.1109/MC.2024.3390948.

Bordis, Tabea, Tobias Runge, Fynn Demmler, and Ina Schaefer (2025 (Submitted)). “Improving Correctness-by-Construction Engineering by Increasing Levels of Correctness Guarantees”. In: *International Journal on Software Tools for Technology Transfer*.

Bordis, Tabea, Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer (Feb. 2020a). “Variational Correctness-by-Construction”. In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. Magdeburg Germany: ACM, pp. 1–9. DOI: 10.1145/3377024.3377038.

Bordis, Tabea, Tobias Runge, and Ina Schaefer (Nov. 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*. Virtual USA: ACM, pp. 22–34. DOI: 10.1145/3425898.3426959.

Bordis, Tabea, Tobias Runge, David Schultz, and Ina Schaefer (June 2022b). “Family-Based and Product-Based Development of Correct-by-Construction Software Product Lines”. In: *Journal of Computer Languages* 70, p. 101119. DOI: 10.1016/j.cola.2022.101119.

Publications Related to This Thesis

Bordis, Tabea and K. Rustan M. Leino (2024). “Free Facts: An Alternative to Inefficient Axioms in Dafny”. In: *Formal Methods*. Vol. 14933. Cham: Springer Nature Switzerland, pp. 151–169. DOI: 10.1007/978-3-031-71162-6_8.

Bordis, Tabea, Tobias Runge, Alexander Kittelmann, and Ina Schaefer (Apr. 2023b). “Correctness-by-Construction: An Overview of the CorC Ecosystem”. In: *ACM SIGAda Ada Letters* 42.2, pp. 75–78. DOI: 10.1145/3591335.3591343.

Kittelmann, Alexander, Tobias Runge, Tabea Bordis, and Ina Schaefer (2022). “Runtime Verification of Correct-by-Construction Driving Maneuvers”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*. Vol. 13701. Cham: Springer International Publishing, pp. 242–263. DOI: 10.1007/978-3-031-19849-6_15.

Kodetzki, Maximilian, Tabea Bordis, Michael Kirsten, and Ina Schaefer (2025a). “Towards AI-Assisted Correctness-by-Construction Software Development”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering Methodologies*. Vol. 15222. Cham: Springer Nature Switzerland, pp. 222–241. DOI: 10.1007/978-3-031-75387-9_14.

Kodetzki, Maximilian, Tabea Bordis, Alex Potanin, and Ina Schaefer (2025b). “X-by-Construction: Towards Ensuring Non-functional Properties in By-Construction Engineering”. In: *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! ’25)*. Singapore.

Kodetzki, Maximilian, Tabea Bordis, Tobias Runge, and Ina Schaefer (Feb. 2024). “Partial Proofs to Optimize Deductive Verification of Feature-Oriented Software Product Lines”. In: *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems*. Bern Switzerland: ACM, pp. 17–26. DOI: 10.1145/3634713.3634714.

Kuiter, Elias, Alexander Knüppel, Tabea Bordis, Tobias Runge, and Ina Schaefer (Feb. 2022). “Verification Strategies for Feature-Oriented Software Product Lines”. In: *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*. Florence Italy: ACM, pp. 1–9. DOI: 10.1145/3510466.3511272.

Rønneberg, Rasmus Carl, Tabea Bordis, Christopher Gerking, Asmae Heydari Tabar, and Ina Schaefer (2025). “Scaling Information Flow Control By-Construction to Component-based Software Architectures”. In: *45th International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Lille, France. DOI: 10.1007/978-3-031-95497-9_4.

Runge, Tobias, Tabea Bordis, Alex Potanin, Thomas Thüm, and Ina Schaefer (June 2023). “Flexible Correct-by-Construction Programming”. In: *Logical Methods in Computer Science* Volume 19, Issue 2, p. 10384. DOI: 10.46298/lmcs-19(2:16)2023.

Runge, Tobias, Tabea Bordis, Thomas Thüm, and Ina Schaefer (2021). “Teaching Correctness-by-Construction and Post-hoc Verification – The Online Experience”. In: *Formal Methods Teaching*. Vol. 13122. Cham: Springer International Publishing, pp. 101–116. DOI: 10.1007/978-3-030-91550-6_8.

2. Background

In this chapter, we briefly introduce the key topics of this thesis. This includes contract-based verification in general (Section 2.1) and Correctness-by-Construction (Section 2.2) as the main approach that we focus on in this thesis. Furthermore, we explain the background of software product line engineering (Section 2.3) and give formal definitions for the core concepts as a basis for our contribution in Chapter 5. Additionally, we introduce a running example that is used throughout this thesis.

2.1. Contract-based Verification

In this thesis, we focus on program verification that is based on contracts which describe properties that a program should have. In Section 2.1.1, we introduce design-by-contract with an example. In Section 2.1.2, we introduce post-hoc verification, which describes the process of specifying and implementing a program first, and afterwards verifying the whole program.

2.1.1. Contracts

Already early on, checking the correctness of programs has been a concern for pioneers in the field of computer science. In 1969, Hoare (1969) proposed the first formal reasoning system, called Hoare logic, with the notation `{P} S {Q}`, called Hoare triple. A Hoare triple guarantees partial correctness and can be interpreted as: Given a program `S`, if the conditions in precondition `P` hold and the program `S` terminates, the conditions in postcondition `Q` hold after termination of `S`. The terms precondition and postcondition, as well as their symbols `P` and `Q` are still commonly used. The Correctness-by-Construction approach that we mainly focus on in this thesis, also builds on these Hoare triples.

The term *contract* has been defined by Liskov and Guttag (1986) and was initially defined by a pre- and postcondition pair. Later, Meyer (1992) proposed *design-by-contract* which generalizes assertions and is used to specify methods in an object-oriented program. A contract consisting of a pre- and postcondition for a method can either guarantee partial or total correctness. Partial correctness means that when the method is called and the precondition is fulfilled, then if it terminates, the postcondition is guaranteed after the method is executed. For total correctness,

```
 1 class IntList{
 2     //@ invariant data != null;
 3     public int[] data = new int[0];
 4 
 5     /*@ requires true;
 6       @ ensures contains(data, newTop) && containsAll(\old(data),
            0, \old(data).length, data);
 7       @ assignable data;
 8       @*/
 9     public void push(int newTop){
10         int[] tmp = new int[data.length+1];
11         tmp[tmp.length-1] = newTop;
12         for(int i = 0; i < data.length; i++) {
13             tmp[i] = data[i];
14         }
15         data = tmp;
16     }
17 
18     ...
19 }
```

Listing 2.1: JML specification of method `push` in class `IntList`.

it also has to be guaranteed that the method terminates. First, design-by-contract was introduced for the programming language EIFFEL (Meyer, 1988), where it was used to define contracts for methods. Furthermore, Hatcliff et al. (2012) reduced the specification effort in EIFFEL by adding syntactic sugar, such as class invariants and framing conditions. Class invariants describe a state that has to be fulfilled before and after executing a method, and frame conditions define which fields can be modified during method execution.

Besides EIFFEL, there are several other specification languages for other programming languages that follow the design-by-contract paradigm, for example, SPEC# (Barnett et al., 2005) for C# programs and JML (Leavens and Cheon, 2006) for JAVA programs. In the following, we present an example of a method `push` in class `IntList` that is specified in JML, as our approach also focuses on JAVA programs, and in some of our evaluations our approach is compared to programs that have been specified with JML and verified in a post-hoc fashion.

Example 2.1. *In Listing 2.1, we show an excerpt of the JAVA code and JML specifications of class* `IntList` *which maintains an integer array* `data` *by offering corresponding methods. We show method* `push` *in detail, which adds a new element* `int newTop` *to the end of array* `data`*. The JML specifications are written as special comments with an @ (Line 2 and Lines 5 – 8). In Line 2, a class invariant is defined. That means that the field* `data` *is not allowed to be* `null` *before and after each method execution. In Lines 5 – 8, the contract of method* `push` *is defined. In Line 5, the precondition with the keyword* `requires` *is defined. In this case,* `true` *is the weakest precondition and means that the postcondition is guaranteed in* *any case (except that* `data` *is* `null`*, as this is required by the class invariant). The postcondition is defined with the keyword* `ensures` *in Line 6. The postcondition uses two predicates. Predicate* `contains(int[] a, int b)` *is* `true` *iff* `b` *is contained in array* `a`*. Predicate* `containsAll(int[] a, int i, int j, int[] b)` *is* `true` *iff the elements of array* `a` *from index* `i` *(inclusive) to index* `j` *(exclusive) are contained in array* `b`*. Therefore, the whole postcondition means that after executing method* `push` *the parameter* `newTop` *is contained in array* `data` *and all old elements of data, i.e., the elements that have been in* `data` *before entering method* `push`*, are still contained in array* `data`*. In Line 7, the framing condition is introduced with the keyword* `assignable` *and defines the set of fields (in this example only field* `data`*) that method* `push` *is allowed to modify.*

Besides method contracts, loops inside a method can also be specified with a loop invariant (Hoare, 1972), a condition that is maintained during loop execution, and a variant (Winskel, 1993), an expression that is decreasing with every iteration of the loop, to show correctness and termination without the need to unroll the loop. To show correctness of a loop, one has to show three conditions. First, the loop invariant must hold before entering the loop. Second, the loop invariant must hold after every iteration. Third, for total correctness, the loop must terminate, which can be shown through the variant that has to decrease monotonically and bounded from below.

2.1.2. Post-hoc Verification

In the previous subsection, we described how methods can be specified by means of behavioral contracts. In this section, we introduce post-hoc verification, which means that verification is performed *after* specification and implementation have been finished. In this thesis, we usually mean deductive verification when we write post-hoc verification, since we compare it to Correctness-by-Construction in our evaluations. However, also model checking and program analysis (Clarke et al., 2018) are prominent contract-based verification techniques that are also applied post-hoc.

Deductive Verification

Theorem proving is a deductive technique to prove the validity of logical formulas. It uses a logical calculus to apply inference rules for conducting a proof mathematically. To prove a program, the program first has to be translated into a logical formula. Then a theorem about the program has to be formulated such that the theorem prover can assist the developer in proving the theorem. Theorem provers can be separated into interactive and automatic theorem provers. Interactive theorem provers, such as Coq (Coq, 1989/2025), Agda (Bove et al., 2009), Lean (De Moura et al., 2015), and Isabelle/HOL (Wenzel et al., 2008), define a tactics language for the inference rules to conduct the proof manually. Automatic theorem provers, such as Vampire (Kovács and Voronkov, 2013) and SPASS (Weidenbach et al., 2009), try to synthesize the proof automatically using a proof-search mechanism. In general, automatic theorem provers are usually restricted to first-order logic while interactive theorem provers focus on higher-order logic, but both of them allow to prove similar problems, including compiler optimization (Leroy, 2009) and real-world programming languages (De Gouw et al., 2015; De Gouw et al., 2019).

Deductive program verifiers take a program P and a contract consisting of a precondition ϕ and a postcondition ψ as input and either prove total or partial correctness. Examples for deductive verifiers are KeY (Ahrendt et al., 2016a) for Java and JML, Frama-C (Cuoq et al., 2012) for C and ACSL, and Viper (Müller et al., 2016) for multiple programming languages. The proof calculus that is used to prove the program consists of a program logic and inference rules to prove theorems. Most of the deductive verifiers use the sequent calculus by Gentzen (1935). A sequent is defined as $\Gamma \vdash \Delta$ where Γ and Δ are sets of formulas. $\Gamma \vdash \Delta$ can also be written as $\bigwedge_{\phi\in\Gamma} \vdash \bigvee_{\psi\in\Delta}$. To prove the correctness of a sequent it has to be shown that this formula is a tautology.

Example 2.2. *The* orLeft *rule of the sequent calculus (Gentzen, 1964):*

$$\frac{\Gamma, \phi \vdash \Delta \qquad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \vee \psi \vdash \Delta}\ (orLeft)$$

The premises above the line ($\Gamma, \phi \vdash \Delta$ and $\Gamma, \psi \vdash \Delta$) are two formulas, which we assume to be tautologies. Therefore, we can assume that also the conclusion ($\Gamma, \phi \vee \psi \vdash \Delta$) of these two premises below the line is a tautology. The symbols Γ and Δ are placeholders for a set of formulas, and $\phi \vee \psi$ is the formula that we want to deduce. For the orLeft *rule, if we have a disjunction on the left side of the turnstile symbol in the conclusion, we split the disjunction into two premises with ϕ/ψ on the left side. We can apply further rules to the two premises. If we have deduced them until we can apply a rule that closes the proof branch, we know that our conclusion is in fact a tautology.*

Program Verification in KeY. KeY is a deductive program verifier for Java programs annotated with JML specifications (Ahrendt et al., 2016a). KeY translates Java programs with JML specifications into formulas in Java Dynamic Logic (JavaDL) (Beckert et al., 2016). JavaDL is an instance of dynamic logic (Harel et al., 2001) tailored to Java, where dynamic logic itself extends first-order logic to reason about programs. To verify that a program P satisfies its contract consisting of precondition ϕ and postcondition ψ, the following dynamic logic formula has to be proven: $\phi \rightarrow \langle P \rangle \psi$. Here, the program modality $\langle P \rangle$ expresses total correctness, i.e., that if precondition ϕ holds and program P terminates, then postcondition ψ will hold after execution. For partial correctness (only if program P terminates, program P will hold), the program modality $[P]$ is used.

The verification process in KeY starts with translating the JML annotated Java implementation into a JavaDL formula. All JML `requires` clauses are composed into one precondition, and all JML `ensures` clauses are composed into one postcondition. Implicit assumptions, such as fields that cannot be `null`, are made explicit, and the JAVA heap is modeled explicitly. This results in a JAVADL formula of the form $\phi \rightarrow \{heapAtPre := heap\}\langle self.m(args);\rangle(\psi \land frame)$. Here, precondition ϕ and postcondition ψ are the composed JAVADL pre- and postcondition, variables *heap* and *heapAtPre* is the current and initial modeled heap, and condition *frame* encodes the framing condition translated from the JML `assignable` clause. To prove such a JAVADL formula, KEY uses a sequent calculus based on Gentzen (1964)'s sequent calculus. In Example 2.2, we already showed a sequent calculus rule for first-order logic. To reason about the program behavior in a dynamic formula, KEY uses symbolic execution. Symbolic execution executes the program step-by-step by rewriting the program into updates of the program state (Ahrendt et al., 2016a). In the JavaDL formula above, there is an example for an update $\{heapAtPre := heap\}$ that assigns the current value of the variable *heap* to the variable *heapAtPre*.

2.2. Correctness-by-Construction (CbC)

In contrast to post-hoc verification, *Correctness-by-Construction* (CbC) (Kourie and Watson, 2012) based on the ideas of Morgan (1990), Dijkstra (1976), and Gries (1981) is a more fine-granular approach where specification, implementation, and verification are performed iteratively to create provably correct programs. Every statement is surrounded by a behavioral specification, forming a *Hoare triple* of the form {`P`} `S` {`Q`}. Thereby, the precondition `P` marks the state of the program before the statement `S` is executed and guarantees that the statement will terminate in the state described by postcondition `Q` (total correctness). The precondition `P`, the postcondition `Q`, and the Hoare triple itself are *predicate formulas*, which means that they either evaluate to `true` or `false`. In this thesis, the pre- and postcondition are defined in *first-order logic* and the statements are defined in *Guarded Command Language* (Dijkstra, 1975), using the following five constructs: empty command (*skip*), assignment (:=), composition (;), selection (**if**), and repetition (**do**). Additionally, we extend the language to use method calls in the statements.

When developing a method with CbC, the starting point is a Hoare triple with an abstract statement `S`, whose implementation has not been defined yet. The first step is then to define the pre- and postcondition before implementing anything else. In this thesis, we also refer to the starting triple of a method as its *contract*.

Definition 2.1: Contract c (Bordis et al., 2022b)
We define a contract c in the form of c = {`P`} `m` {`Q`}, with `P` being the precondition, `Q` the postcondition, and `m` $\in \mathbb{M}$ the method. `P` and `Q` are both defined in first-order logic over variables $v \in \mathbb{V}$. We require that each contract $c \in \mathbb{C}$ can be formulated as c = {`P`} `m` {`Q`} regardless of the particular specification language.

Refinement Rules. With the application of *refinement rules* that preserve the correctness at each refinement step, the starting triple can be refined to implement a method. The correctness is preserved because every refinement rule defines *side conditions* that have to be guaranteed when a refinement rule is applied. In Definition 2.2, we present a list of the eight most important refinement rules, where the side conditions of each refinement rule are defined after the *iff*. Generally, a method is correctly implemented when no abstract statement is left.

Definition 2.2: Refinement Rules for Correctness-by-Construction (Kourie and Watson, 2012)
Let $\mathtt{P}$ be the precondition, $\mathtt{Q}$ the postcondition, and $\mathtt{S}$ an abstract statement. Then a Hoare triple $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ can be refined to:

1. **Skip:** $\{\mathtt{P}\}$ *skip* $\{\mathtt{Q}\}$ *iff* $\mathtt{P}$ *implies* $\mathtt{Q}$
2. **Weaken Precondition:** $\{\mathtt{P}'\}\ \mathtt{S}\ \{\mathtt{Q}\}$ *iff* $\mathtt{P}$ *implies* $\mathtt{P}'$
3. **Strengthen Postcondition:** $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}'\}$ *iff* $\mathtt{Q}'$ *implies* $\mathtt{Q}$
4. **Assignment:** $\{\mathtt{P}\}\ \mathtt{x} := \mathtt{E}\ \{\mathtt{Q}\}$ *iff* $\mathtt{P}$ *implies* $\mathtt{Q}[\mathtt{x}\backslash\mathtt{E}]$
5. **Composition:** $\{\mathtt{P}\}\ \mathtt{S}_1\,;\ \mathtt{S}_2\ \{\mathtt{Q}\}$ *iff there is an intermediate condition* $\mathtt{M}$ *such that* $\{\mathtt{P}\}\ \mathtt{S}_1\ \{\mathtt{M}\}$ *and* $\{\mathtt{M}\}\ \mathtt{S}_2\ \{\mathtt{Q}\}$*hold*
6. **Selection:** $\{\mathtt{P}\}$ **if** $\mathtt{G}_1 \rightarrow \mathtt{S}_1$ **elif** $\ldots\ \mathtt{G}_n \rightarrow \mathtt{S}_n$ **fi** $\{\mathtt{Q}\}$ *iff* ($\mathtt{P}$ *implies* $\mathtt{G}_1 \vee \mathtt{G}_2 \vee \ldots \vee \mathtt{G}_n$) *and* $\{\mathtt{P} \wedge \mathtt{G}_i\}\ \mathtt{S}_i\ \{\mathtt{Q}\}$ *holds for all* $\mathtt{i}$
7. **Repetition:** $\{\mathtt{P}\}$ **do** $[\mathtt{I}, \mathtt{V}]\ \mathtt{G} \rightarrow \mathtt{R}$ **od** $\{\mathtt{Q}\}$ *iff* ($\mathtt{P}$ *implies* $\mathtt{I}$) *and* ($\mathtt{I} \wedge \neg\mathtt{G}$ *implies* $\mathtt{Q}$) *and* $\{\mathtt{I} \wedge \mathtt{G}\}\ \mathtt{S}\ \{\mathtt{I}\}$ *holds and* $\{\mathtt{I} \wedge \mathtt{G} \wedge \mathtt{V}{=}\mathtt{V}_0\}\ \mathtt{S}\ \{\mathtt{I} \wedge 0{\leq}\mathtt{V} \wedge \mathtt{V}{<}\mathtt{V}_0\}$ *holds*
8. **Method Call:** $\{\mathtt{P}\}\ \mathtt{b} := \mathtt{m}(\mathtt{a}_1, ..., \mathtt{a}_n)\ \{\mathtt{Q}\}$ *iff method* $\mathtt{m}$ *with signature and contract* $\{\mathtt{P}'\}\ \mathtt{return\ r\ m(param}\ \mathtt{p}_1, ..., \mathtt{p}_n)\ \{\mathtt{Q}'\}$ *exists and* $\mathtt{P}$ *implies* $\mathtt{P}'[\mathtt{p}_i\backslash\mathtt{a}_i]$ *and* $\mathtt{Q}'[\mathtt{p}_i^{old}\backslash\mathtt{a}_i^{old}, \mathtt{r}\backslash\mathtt{b}]$ *implies* $\mathtt{Q}$

Skip. The skip rule replaces an abstract statement $\mathtt{S}$ with the skip command, that does not alter the program state. Hence, it is applicable iff the precondition $\mathtt{P}$ implies the postcondition $\mathtt{Q}$ (i.e., iff the program is already in a state that satisfies postcondition $\mathtt{Q}$).

Weaken Precondition. The precondition $\mathtt{P}$ can be replaced by a weaker condition $\mathtt{P}'$, iff $\mathtt{P}$ implies $\mathtt{P}'$. In other words, the weaker precondition $\mathtt{P}'$ is sufficient to guarantee the same postcondition $\mathtt{Q}$.

Strengthen Postcondition. The postcondition $\mathtt{Q}$ can be replaced by $\mathtt{Q}'$, iff $\mathtt{Q}'$ is stronger (i.e., $\mathtt{Q}'$ implies $\mathtt{Q}$).

Assignment. An abstract statement $\mathtt{S}$ can be replaced by an assignment $\mathtt{x} := \mathtt{E}$ iff precondition $\mathtt{P}$ implies postcondition $\mathtt{Q}$ in which the variable $\mathtt{x}$ has been replaced by an expression $\mathtt{E}$. We refer to this replacement with the notation $\mathtt{Q}[\mathtt{x} \backslash \mathtt{E}]$. Multiple assignments can be expressed in one statement as $\mathtt{x_0}, \mathtt{x_1} := \mathtt{E_0}, \mathtt{E_1}$, where $\mathtt{E_0}$ is the assignment for $\mathtt{x_0}$ and $\mathtt{E_1}$ is the assignment for $\mathtt{x_1}$.

Composition. The composition rule splits one abstract statement into two abstract statements $\mathtt{S_1}$ and $\mathtt{S_2}$. Additionally, an intermediate condition $\mathtt{M}$ has to be provided, such that both $\{\mathtt{P}\}\mathtt{S_1}\{\mathtt{M}\}$ and $\{\mathtt{M}\}\mathtt{S_2}\{\mathtt{Q}\}$ hold. The Hoare triples $\{\mathtt{P}\}\mathtt{S_1}\{\mathtt{M}\}$ and $\{\mathtt{M}\}\mathtt{S_2}\{\mathtt{Q}\}$ can both be further refined by applying refinements rules afterwards.

Selection. The selection rule introduces a selection where multiple conditional statements $\mathtt{S_i}$ are refined that are only executed if their guard $\mathtt{G_i}$ is true. The Hoare triple is refined to n more Hoare triples of the form $\{\mathtt{G_i} \land \mathtt{P}\}\mathtt{S_i}\{\mathtt{Q}\}$. The guards $\mathtt{G_i}$ are evaluated and the substatement of the first satisfied guard is executed. The side condition of the selection rule is that precondition $\mathtt{P}$ must imply the disjunction of all guards, such that it can be guaranteed that at least one guard evaluates to true.

Repetition. The repetition refinement rule introduces a loop with a statement $\mathtt{S}$ that is executed as long as guard $\mathtt{G}$ evaluates to `true`. To guarantee the conformance to the postcondition and termination, an invariant $\mathtt{I}$ and a variant $\mathtt{V}$ have to be defined. The invariant is a condition that is `true` before and after each iteration of the loop. The variant $\mathtt{V}$ is an integer expression that decreases monotonically with each iteration and has zero as lower bound. The side condition of the repetition rule consists of four conditions: 1) Invariant $\mathtt{I}$ has to imply precondition $\mathtt{P}$ to ensure that the invariant I hold before entering the loop. 2) Invariant $\mathtt{I}$ conjoined with the negated guard $\mathtt{G}$ has to imply the postcondition $\mathtt{Q}$, to guarantee that the postcondition is satisfied after leaving the loop. 3) The loop body $\mathtt{R}$ must preserve the invariant $\mathtt{I}$. 4) The termination of the repetition construct is verified by showing that the variant $\mathtt{V}$ is monotonically decreasing with zero as lower bound. The expression $\mathtt{V_0}$ denotes to the value of the variant $\mathtt{V}$ before a loop iteration.

Method Call. The method call refinement rule calls a method $\mathtt{m}$ that is defined with a contract and the following signature: $\{\mathtt{P'}\}$ `return r m(param` $\mathtt{p_1, ..., p_n}$`)` $\{\mathtt{Q'}\}$. Thereby, the formal parameters $\mathtt{p_1, ..., p_n}$ have a scope that is limited to the method body, and the return variable $\mathtt{r}$ gets assigned a new value after the execution of method $\mathtt{m}$. Both, the formal parameters $\mathtt{p_1, ..., p_n}$ and the return variable $\mathtt{r}$ can also be empty. To omit side effects in all cases, we define the method parameters as call-by-value. As a result, the method call rule can generally be applied if the specification of the method complies with the specification of the statement. That is checking that the precondition of the method call refinement $\mathtt{P}$ implies the precondition of the method $\mathtt{P'}$ and that the postcondition of the method $\mathtt{Q'}$ implies the postcondition of the method call refinement $\mathtt{Q}$, but with the formal parameters $\mathtt{p_1, ..., p_n}$ and return value $\mathtt{r}$ replaced by their actual values in the method call $\mathtt{a_1, ..., a_n}$ and $\mathtt{b}$. Because of the limited scope of the formal parameters, they are not allowed to appear in the postcondition $\mathtt{Q'}$. However, $\mathtt{Q'}$ may refer to their value before executing the method, which we denote as

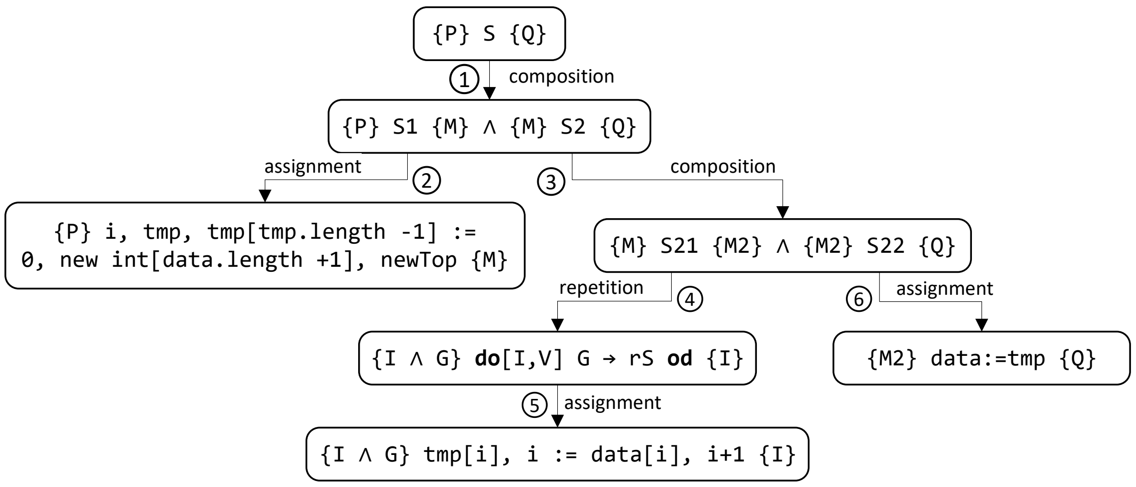

{P} S {Q}
1
composition
{P} S1 {M} ∧ {M} S2 {Q}
assignment
2
3
composition
{P} i, tmp, tmp[tmp.length -1] := 0, new int[data.length +1], newTop {M}
{M} S21 {M2} ∧ {M2} S22 {Q}
repetition
4
6
assignment
{I ∧ G} do[I,V] G → rS od {I}
{M2} data:=tmp {Q}
5
assignment
{I ∧ G} tmp[i], i := data[i], i+1 {I}

Figure 2.1.: CbC refinement tree for method `push` (Bordis et al., 2020a). Java code and JML specification for method `push` are shown in Listing 2.1.

$\mathtt{p}^{\mathtt{old}}$. The return value is not allowed to appear in the precondition, as it does not exist before executing the method.

2.2.1. Correctness-by-Construction by Example

In the following, we demonstrate how CbC is applied to develop correct programs at the example of the `push` method that we have already used as an example for a JML contract in Listing 2.1. To start development, we first define the pre- and postconditions. The reason behind this is that CbC enforces a think-first process, where the intent should be formalized before implementing anything. In our case, we set the precondition `P` to `true` which is the weakest precondition. To simplify the postcondition, we use the predicates `contains(int[] a, int b)` and `containsAll(int[] a, int i, int j, int[] b)`. Thereby, the predicate `contains` evaluates to `true`, iff array `a` contains value `b` and `containsAll` evaluates to `true`, iff array `a` contains all elements of array `b` from index `i` (inclusive) to `j` (exclusive). In our example, the postcondition ensures that array `data` contains the new element `newTop` and all elements that it has contained before the execution of the method `push`. The concrete postcondition using our self-defined predicates is defined as follows: $\mathtt{contains(data,\ newTop)} \wedge \mathtt{containsAll(data,\ 0,\ data.length,\ data}^{\mathtt{old}}\mathtt{)}$.

To implement method `push`, we want to create a temporal array `tmp`, which shall contain all elements from `data` and additionally the new element `newTop`. Afterwards, we want to assign `tmp` to `data` to finalize the algorithm. In a first step, we apply the *composition refinement rule* ① to split the abstract statement `S` into two abstract statements. As we need to traverse through array `data` later on in the algorithm, we already know that we need a loop counter variable. Additionally, `tmp` needs to be created with the length of `data` plus 1 and we can already add `newTop` as well. We can formalize these requirements in the intermediate condition `M` as follows: `tmp.length = data.length+1` ∧ `i = 0` ∧ `contains(tmp, newTop)`.

To fulfill this intermediate condition, we apply the *assignment refinement rule* ② to the first abstract statement `S1`. We define three assignments. First, we assign `0` to the loop counter variable `i`. Second, we create the array `tmp` with the length `data.length+1`. Third, we assign `newTop` to the last index of `tmp`.

For the second abstract statement `S2` of the composition statement, we need to add every element of `data` to `tmp` and as a last step of the algorithm, assign `tmp` to `data`. Consequently, we need to split the second abstract statement `S2` again using the *composition refinement rule* ③. As the assignment of `tmp` to `data` shall be the last step, we define the intermediate condition `M2` as follows: `tmp.length = data.length+1 ∧ contains(tmp, newTop) ∧ containsAll(tmp, 0, data.length, data)`. The array `tmp` shall contain `newTop` and all elements from `data`, as these are the conditions that need to be fulfilled for `data` after the assignment as well.

As a next step, we further refine the first abstract statement of the second composition (`S21`). The parts of the intermediate condition `M2` `contains(tmp, newTop)` and `tmp.length = data.length+1` are already fulfilled by `tmp`. Therefore, we only need to fulfill the remaining part of `M2`, which is `containsAll(tmp, 0, data.length, data)`. This can be achieved by applying the *repetition refinement rule* ④. It requires the additional definition of a guard, an invariant, and a variant to guarantee the correct postcondition and the termination of the loop. We select `i < data.length` as the guard, as we want to traverse `data` from left to right. The variant can be chosen as `data.length - i` using the loop counter variable `i`, which is monotonically decreasing in the loop. Finally, we define an invariant for the loop. We use the conditions `tmp.length = data.length+1 ∧ contains(tmp, newTop)` from the intermediate condition `M` and add the condition `containsAll(tmp, 0, i, data)`, which reflects the behavior of adding each element from `data` to `tmp` during the repetition of the loop. To implement the loop body, we refine the abstract repetition statement **`rS`** using the *assignment refinement rule* ⑤. We assign the value of `data` at index `i` to `tmp` at the same index and increase `i` by 1.

To finalize the algorithm, we need to concretize the last abstract statement `S22` by assigning `tmp` to `data` using again the *assignment refinement rule* ⑥. After checking the side condition of this assignment refinement step ⑥, we know that all refinement steps preserve the correctness of the initial pre- and postcondition `P` and `Q` and that therefore method `push` is functionally correct w.r.t. this specification.

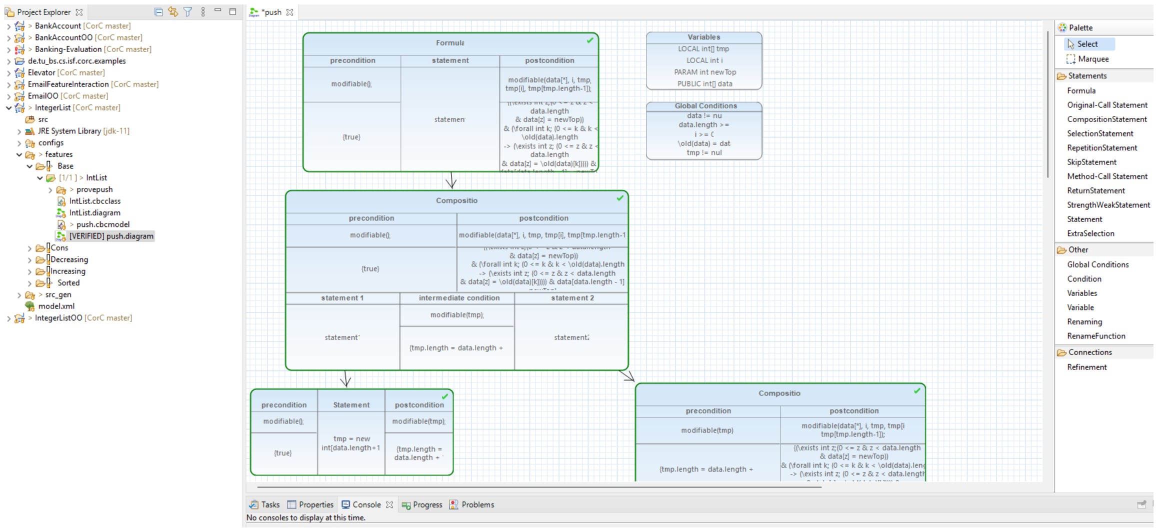


Figure 2.2.: Graphical editor of CorC displaying the push method from Figure 2.1 and Listing 2.1.

2.2.2. Tool Support: CorC

CorC (Runge et al., 2019) is an open-source Eclipse plug-in[1] that supports the development of correct software systems with CbC. In Figure 2.2, we show a screenshot of CorC. The user interface of CorC consists of a project explorer and a graphical editor for development. On the right-hand side of the graphical editor, there is a list of refinement rules that can be added to the program via drag and drop. The constructed programs are saved as instances of an Ecore metamodel [2] that internally represents the structure of CbC programs.

Verification of Refinement Steps with KeY. The correctness of each refinement step is checked with the deductive program verifier KeY (Ahrendt et al., 2016a) by establishing that the side condition of that refinement rule is met (see Definition 2.2). CorC translates these side conditions together with the concrete statements and specifications into a proof obligation, where the code is written in Java syntax with specifications in JavaDL (Ahrendt et al., 2016a). The language constructs covered by CorC are similar to the guarded command language (Dijkstra, 1975) with statements for skip, assignment, method call, composition, selection, and repetition. The whole verification process with KeY is automated. In CorC, we normally only receive the verification result from KeY. First, a proof file with the proof obligation and settings for KeY is generated automatically. This proof file together with the whole project is loaded by KeY. At this point, KeY checks the proof file for syntactical errors and throws self-defined exceptions. If there are no syntactical errors, KeY applies the rules of its calculus to verify the side condition in the proof file. All applied proof

[1] CorC is available at: https://github.com/KIT-TVA/CorC
[2] https://eclipse.dev/modeling/emf/

steps are saved in the proof file that we initially generated, so that the CorC user can analyze the proof steps later on. KeY returns the verification result to CorC, and the border of the refinement step is marked green if the proof could be closed and red otherwise.

In comparison to post-hoc verification of whole Java methods in KeY, CorC splits the verification task into the verification of several refinement steps (checking the refinements of introducing skip, assignment, method call, composition, selection, repetition, weakening precondition, and strengthening postcondition). In each refinement step, a side condition, such as the establishment of a loop invariant before the first loop iteration, is verified. All proofs combined guarantee the correctness of the whole method. This split into several proofs can reduce the proof complexity and increase the proof efficiency (Watson et al., 2016). Thus, CorC can be used as a frontend for KeY that enables CbC development for the construction of individual algorithms.

2.3. Software Product Line Engineering

A *software product line* is a set of software products as variations of a common code base (Cadar and Nowack, 2021). Software artifacts are systematically reused using variable code structures so that these artifacts can be composed according to the needs of the customer. Thus, product lines allow implementing a whole product family more efficiently than multiple single software products in isolation. Software product line engineering consists of two phases, called *domain engineering* and *application engineering* (Pohl et al., 2005). In domain engineering, the commonalities and differences of the product variants in the family are defined and realized in terms of *features*. The definition of features often varies in the literature depending on the viewpoint (Classen et al., 2008). We simply define features as unique labels with no further structure (Bordis et al., 2022b). In application engineering, a feature configuration is created adhering to the restrictions of the feature model. Afterwards, in the domain engineering phase, the artifacts for the selected features are composed with a corresponding software product line realization technique to generate the desired product variant of the product line.

The features of a product line are usually organized in *feature models* (Kang et al., 1990). Features can be defined as mandatory, optional, or collected in groups, which leads to dependencies that are often captured in tree structures. For simplicity, we refer to a feature model as the set of all valid *feature configurations*. A valid feature configuration is a sequence of features adhering to the dependencies of the feature model. The sequential order is defined by the *feature composition order* that can be statically defined by the feature model.

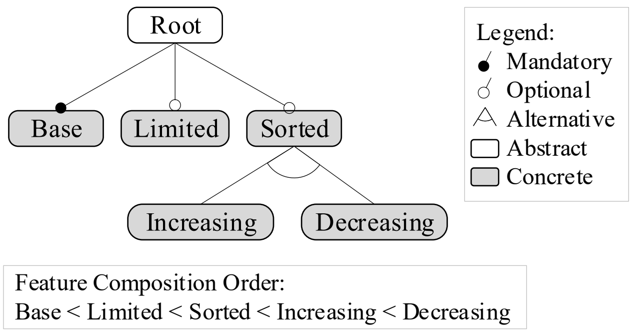


Figure 2.3.: Feature model of *IntegerList* product line (Bordis et al., 2022b).

Definition 2.3: Feature Model FM (Bordis et al., 2022b)
Let $\mathbb{F}$ be the universe of feature labels. A feature model FM is a set of valid feature configurations $\text{FM} \subseteq \mathcal{P}(\mathbb{F})$ and a feature composition ordering $<$. The feature composition order is defined as a partial order $\{(f_1, f_2) \in \mathbb{F}^2\}$ which we write as $f_1 < f_2$. A valid feature configuration including n features is a sequence of these features $[f_1, ..., f_n]$ such that for $i < j : f_i < f_j$.

Example 2.3. *We introduce the IntegerList as an exemplary product line implementation. The IntegerList maintains an integer array* `data` *by offering corresponding methods. We show the feature model of the IntegerList product line in Figure 2.3. The method* `push` *adds a new integer* `newTop` *to the array in the implementation of feature Base. This is the implementation that we have already shown in Listing 2.1. Feature* Limited *limits the length of array* `data` *so that* `newTop` *is only added if the limit is not reached yet. Feature* Sorted *ensures that the array* `data` *is always sorted, either in an increasing or a decreasing order (implemented by sub-features* Increasing *and* Decreasing*). Even for this rather small example, there are in total six valid configurations (*$\text{FM} = \{[Base], [Base, Limited], [Base, Sorted, Increasing], [Base, Sorted, Decreasing], [Base, Limited, Sorted, Increasing], [Base, Limited, Sorted, Decreasing]\}$*).*

Based on our definition of a feature model, we define partial configurations and the set of valid configurations for a distinct feature to reason about the configurations that have a specific feature selected as follows.

Definition 2.4: Partial Configuration *pfc* (Bordis et al., 2022b)
A feature configuration *pfc* is a *partial configuration* of feature model FM iff there exists $fc \in \text{FM}$ such that $pfc \subseteq fc$. We define the predicate *isPartialConfig(*FM, *pfc)* which evaluates to `true` iff *pfc* is a partial configuration of feature model FM.

Definition 2.5: Feature Configurations for Feature f FM$_f$ (Bordis et al., 2022b)
Let $\mathrm{FM}_f = \{fc \in \mathrm{FM} \mid f \in fc\}$ be the set of valid feature configurations of feature model FM that contain feature f.

Example 2.4. *The feature configuration fc = [*Base, Sorted*] is a partial configuration of feature model* FM *from the IntegerList because there exists at least one valid feature configuration that contains both features (e.g., [Base, Sorted, Increasing] or [Base, Limited, Sorted, Decreasing]). A counter example is the feature configuration fc2 = [Increasing, Decreasing] which is not a partial configuration of feature model* FM.
The set of valid feature configurations for feature Sorted of the IntegerList feature model is defined as $\mathrm{FM}_{Sorted} = \{[Base, Sorted, Increasing], [Base, Sorted, Decreasing], [Base, Limited, Sorted, Increasing], [Base, Limited, Sorted, Decreasing]\}$.

In some concepts in this thesis, we use *presence conditions*, which are propositional logic formulas defined over the features of the feature model. To evaluate whether a presence condition is fulfilled for a certain feature configuration, the features used in the presence condition have to be evaluated to `true` or `false`. Every feature that is contained in that feature configuration is evaluated to `true` and every feature that is not contained is evaluated to `false` respectively. Since a presence condition is defined solely over the features of the feature model, the whole presence condition can be evaluated to `true` or `false` afterwards. Formally, we define the valuation of a feature configuration as follows:

Definition 2.6: Valuation of a Feature Configuration v_{fc}
Let $\mathbb{F}$ be the universe of feature labels and fc be a feature configuration of feature model FM. Then we define the valuation of a feature configuration fc as $v_{fc} : \mathbb{F} \to \{true, false\}$ as follows:

$$v_{fc}(f) = \begin{cases} true, & \textit{iff } f \in fc \\ false, & otherwise \end{cases}$$

Software product line realization techniques. The variability that has been modeled on the domain level also has to be implemented on the code level. As software product lines aim to implement whole product families, features are implemented once and are composed for the desired feature configuration and ordering. Software product line realization techniques can be distinguished into *annotation-based* and *compositional* realization techniques (Apel et al., 2013a).

Annotation-based realization techniques provide a virtual separation of features, which means that every feature is implemented in one single code base. The separation comes in the form of annotations that mark code fragments belonging to one feature. To create a specific product variant from the product family, the code of deselected features is either ignored or removed (Apel et al., 2013a). An example for an annotation-based realization technique is the C-preprocessor (Gacek and Anastasopoules, 2001).

While annotation-based realization techniques provide a virtual separation of features, the compositional realization techniques offer a physical separation. Physical separation means that all artifacts belonging to a certain feature are modularized into one cohesive unit (Apel et al., 2013a), called *feature module*. During product derivation, the units of all features in a feature configuration are composed. The functionality of this composition depends on the concrete software product line realization technique and is defined in the following for feature-oriented programming.

Feature-Oriented Programming

For our contribution in Chapter 5, we prefer a compositional realization technique, as it separates the implementation into smaller units stored in the feature modules which offers a separated structure compared to annotation-based realization techniques. In general, a feature module is not restricted to containing code artifacts. It can possibly contain files of any type, such as images or text files. However, the focus of this thesis is the correct development of methods in a product line, and therefore our definition of feature modules assumes a set of methods as the implementation of a feature module. In this context, a method is a single function or procedure with a signature (return type, name, and a list of parameters) that implements a functionality. Methods are contained in a class that can also have fields, however, we assume that all fields and methods are globally accessible.

Definition 2.7: Feature Module of Feature f impl(f)[a] (Bordis et al., 2022b)
Let $\mathbb{M}$ be the universe of methods. Then we define the feature module of a feature f as a set of methods *impl(f)* $\subseteq \mathbb{M}$. One method can be contained in more than one feature module of the product line.

[a] For brevity, we use the wording that a *feature implements a method* instead of writing that *a feature module of a feature contains that method* in the non-formal written parts of this thesis.

Example 2.5. *For the IntegerList product line, there is a feature module for each feature in the feature model* FM, *i.e., Base, Limited, Sorted, Increasing, and Decreasing. The feature modules contain the following methods:*

- impl(*Base*) = {`void push(int newTop), int pop()`}
- impl(*Limited*) = {`void push(int newTop)`}
- impl(*Sorted*) = {`void push(int newTop), int pop()`}
- impl(*Increasing*) = {`int[] sort(int[] a)`}

- impl(*Decreasing*) = {`int[] sort(int[] a)`}

In this thesis, we focus on *feature-oriented programming* (Apel et al., 2013a; Batory et al., 2004) as a specific compositional technique to realize CbC product lines (Chapter 5). In feature-oriented programming, the implementation of one method can be separated into multiple feature modules for several features. Thereby, the different implementations of this method override each other in the feature composition ordering from smallest to greatest as defined in the feature model. However, an `original` call can be used to call the implementation of the method in the previous feature modules. This general mechanism is similar to overriding with super-calls in object-oriented languages like JAVA. We define method refinement in feature-oriented programming as follows.

Definition 2.8: Method Refinement (Bordis et al., 2022b)
Let $\mathtt{m} \in \mathbb{M}$ be a method and $f_1, ..., f_n \in \mathbb{F}$ the features occurring in the set of valid feature configurations of feature model FM for which $\mathtt{m} \in \text{impl}(f_1), ...,$ $\text{impl}(f_n)$ applies.[a] The feature composition ordering of FM is defined as $f_1 < ... < f_n$. Then the smallest feature containing m (impl(f_1)) introduces the method. All other features *refine* this method by overriding it from smallest to greatest feature according to the feature composition ordering.

[a] Note that there may be other features than $f_1, ..., f_n$ that do not contain method m. These are not relevant for this definition.

Because of the `original` call, the implementation of one method can vary for different feature configurations. The distinct implementation for one feature configuration is formed by overriding the method in the feature modules of the features from the configuration from smallest to greatest. `Original` calls are resolved with a call to the method implementation of the next smaller feature accordingly.

Example 2.6. *In Listing 2.2, we show the implementations of method* `push` *in the feature modules of the features* Limited *and* Sorted*. The implementation of method* `push` *of feature* Base *has already been shown in Listing 2.1. As programming language, we use* FEATUREHOUSE *(Apel et al., 2013b) which is an extension of* JAVA *for feature-oriented programming. The implementation in feature module* Base *creates a local array* `tmp` *with the length of array* `data` *increased by one. Then the new element* `newTop` *is inserted at the last index of* `tmp`*, and all elements in* `data` *are inserted using a for loop. In the end, array* `tmp` *is assigned to* `data`*. The implementation of method* `push` *in feature module* Limited *uses a selection statement, such that the new element* `newTop` *is only inserted if the length of array* `data` *has not reached the value of the limit. In this selection statement, the feature-oriented programming-specific keyword* `original` *is used to call the implementation of method* `push` *in the next smaller feature module according to the feature composition order, which is in this case the implementation in feature module* Base*. The implementation in feature module* Sorted *also uses the keyword* `original` *to first add the new element* `newTop` *to array*

```
class IntList{                                           Limited
  public int MAX_LENGTH = 10;

  public void push(int newTop){
    if(data.length < MAX_LENGTH){
      original(newTop);
    }
  }
}
```

```
class IntList{                                           Sorted

  public void push(int newTop){
    original(newTop);
    data = sort(data);
  }

  public int pop(){...}
}
```

Listing 2.2: Implementation of method `push` in different feature modules using an `original` call (Bordis et al., 2022b).

`data` *and afterwards sorts it by calling method* `sort`*. The implementation of method* `sort` *is not shown in this listing, as it is defined in feature modules* Decreasing *and* Increasing *(see Example 2.5). Depending on the distinct feature configuration, the* `original` call *either refers to the implementation of* `push` *in feature module* Limited *or* Base*. For a feature configuration* $fc = [Base, Limited, Sorted, Decreasing]$ *the* `original` *call refers to the implementation in feature module* Limited *as it is the next smaller feature according to the feature composition ordering. However, for a feature configuration* $fc2 = [Base, Sorted, Decreasing]$ *it directly refers to the implementation of feature* Base *since* Limited *is not included.*

Besides the `original` call, which is resolved differently depending on the feature configuration, also normal method calls can introduce variability in a method. This happens when a called method is defined in more than one feature module and therefore its implementation is also dependent on the distinct feature configuration. We use the term *variational method call* for a method call referring to a method with varying implementations in the product line. For example, the implementation of method `push` in feature *Sorted* calls method `sort` which is a variational method call as method `sort` is defined in more than one feature module (*Increasing* and *Decreasing*) and therefore the implementation varies depending on whether *Increasing* or *Decreasing* is contained in a feature configuration.

Part II.

Contributions to Scale Correctness-by-Construction

3. CorC 2.0: Introducing Objects and an Integrated Software Development Process to CbC

The content of this chapter is largely based on the paper "Re-CorC-ing KeY: Correct-by-Construction Software Development Based on KeY" (Bordis et al., 2022a).

CbC targets the development of single and provably correct algorithms. Today, object-orientation is supported by most modern programming languages and is therefore one of the core paradigms in software engineering. The main benefits of object-orientation are code organization through encapsulation, reusability through inheritance, flexibility through polymorphism, and simplification of complex systems through abstraction. Consequently, object-orientation allows the development of modular, maintainable, and scalable software systems. It also forms the basis for further concepts, such as distributed systems (Briot et al., 1998), architectural design (Gamma et al., 1995), model-based software engineering (Brambilla et al., 2017), and software product line engineering (Apel et al., 2013a).

While integrating object-oriented structures into CbC development increases the size of systems that can be constructed, seamless integration into existing software development processes is essential for practical adoption by developers. As CbC is a formal method, its application domains are mainly safety-critical systems, such as cars, airplanes, or medical devices. These systems can be large and complex, comprising both safety-critical and non-safety-critical components, each requiring different quality assurance techniques. In the automotive sector, the assignment of safety levels is included in ISO 26262-1:2018 (2018) for functional safety of road vehicles. For example, the HMI of a car may need to be tested, while the brake may need to be verified. Moreover, different quality assurance techniques and tools have different advantages and disadvantages. Comparing CbC with post-hoc verification, on the one hand, there is evidence that CbC in CorC has a higher proof efficiency compared to post-hoc verification in KeY (Runge et al., 2019; Runge et al., 2021) and that the incremental, specification-guided process of CbC helps developers to design more efficient algorithms while keeping the structure maintainable and verifiable (Kourie and Watson, 2012). On the other hand, when using post-hoc verification, the specification effort might be lower than with CbC, and the general approach is closer to programming, which might be more intuitive for software developers (Runge et al., 2020). However, even if a system has been formally specified and verified using CbC or post-hoc verification, integration and system tests still need to be performed to ensure that individual components, including third-party components that cannot be verified, work together as intended. Based on these findings, we believe that CbC might be a good approach to develop challenging algorithms inside a software system where also post-hoc verification is applied for other parts of the system, and again, other parts might be tested. Overall, we envision an integrated process that enables combined use of CbC development with other quality assurance techniques, such as post-hoc verification and testing. This is essential to 1) develop individual safety-critical components using CbC, 2) ensure compatibility with existing tool chains, and 3) leverage the strengths of different quality assurance techniques and tools.

In this chapter, we present CorC 2.0, which incorporates key object-oriented concepts – encapsulation, inheritance, polymorphism, and dynamic dispatch – alongside a roundtrip engineering process that allows CbC to be integrated with other quality assurance techniques, such as software testing and post-hoc verification. With this contribution, we address Challenge 1 (scalability to larger systems) and Challenge 2 (integrability into existing software development processes) and answer Research Question 1: *How can we scale Correctness-by-Construction to object-oriented programs, and how can we incorporate Correctness-by-Construction into an integrated software development process?* First, we extend CbC with core object-oriented features, such as classes with invariants, fields, methods, and interfaces. Second, we propose a three-step roundtrip engineering process that allows CbC to be integrated with any quality assurance technique that can be applied to Java code in a textual IDE (e.g., unit testing or post-hoc verification with KeY). We combine both concepts in CorC 2.0, an extension of CorC. We use CorC 2.0 to compare CbC with post-hoc verification in KeY regarding proof efficiency (i.e., the proof nodes and verification time needed to prove a proof obligation) and usability. The results of the proof efficiency evaluation do not indicate that one of these two approaches, CbC or post-hoc verification, is universally superior. In the two user studies that we summarize for the usability evaluation, the participants liked the refinement-based development style, but post-hoc verification was more familiar. These observations underline our initial hypothesis that a universally superior quality assurance technique for all subject systems does not exist. Instead, a collaborative use of different quality assurance techniques, such as CbC, post-hoc verification, and testing, is essential to leverage the advantages of each technique.

This chapter is structured as follows: In Section 3.1, we introduce the conceptual part of this chapter, consisting of object-oriented concepts for CbC in Section 3.1.1 and an integrated development process in Section 3.1.2. In Section 3.1.3, we show the implementation of the concepts in CorC's successor, namely CorC 2.0. In Section 3.2, we evaluate our concept as implemented in CorC 2.0 in terms of its verification effort and usability compared to post-hoc verification. Finally, we present related work in Section 3.3 and summarize this chapter in Section 3.4.

3.1. Object-oriented Development in CorC 2.0

In Section 2.2, we described how single algorithms can be created using CbC in CorC. However, these algorithms are independent of any class structure, which means that these single algorithms cannot access the same set of global fields as methods in a class in object-oriented programming can. Additionally, classes where certain methods shall be created with CbC in CorC, cannot directly integrate the implementation in form of a CbC refinement tree, but the user has to manually transform the CbC refinement tree into Java code. In other words, the current implementation of CorC is difficult to integrate into a software development process as it lacks concepts for modularization of objects and classes, as well as processes that allow CbC to be integrated into existing software development workflows.

Therefore, in this section, we present our concept for CbC-based object-oriented software development and the corresponding extension of CorC in the tool CorC 2.0. CorC 2.0 implements a roundtrip engineering from existing Java projects to CbC-based program development, allowing for a combination of CbC-based program development and other quality assurance techniques, such as testing and post-hoc verification. In the following, we describe the core concepts that we use to integrate object-orientation in CorC 2.0 and the roundtrip engineering process that allows CorC 2.0 to be used in combination with other verification or testing tools.

3.1.1. Object-oriented Concepts in CorC 2.0

Object-oriented programming is a common programming paradigm based on objects, which contain data fields and methods. In class-based languages, such as Java, C++, C#, PHP, or Smalltalk, objects are instances of classes that must be defined in advance. Classes are extensible templates for creating objects, providing initial values for fields and implementations of methods. Other paradigms shared by most object-oriented languages are encapsulation, inheritance, polymorphism, and dynamic dispatch. Since CorC already uses Java syntax in the refinement steps and KeY as a deductive verification tool to verify the single refinement steps, we focus on object-orientation as realized in Java and how this can be combined with CbC.

Classes. To support object-orientation in CorC 2.0, we introduce the construction of classes that contain methods implemented with CorC and fields that can be accessed by the methods contained in the respective class. The visibility of fields can be modified using the Java visibility modifiers `public`, `private`, `package`, and `protected`. Fields can also be defined as `static` or `final`. Methods in a class can define an additional set of local variables, including parameters and a return variable. We also add class invariants as a new type of specification to our class definitions. Class invariants specify conditions that are satisfied by the class, i.e., that are preserved by all methods of that class or re-established at the end of method execution. To ensure that a method created with CorC fulfills the class invariants, they are automatically added to the pre- and postcondition of the starting Hoare triple when the method is constructed.

Inheritance and Interfaces. Inheritance and interfaces are two important features of Java. While interfaces can be used as an abstraction layer, inheritance can be used to create classes based on existing classes, for example, to enable code reuse. To ensure that polymorphism can be correctly applied to both inheritance and interfaces, we check that the Liskov principle is satisfied by the child class or the class that implements an interface. This means that class invariants that are defined in a parent class or an interface must also be fulfilled by the class that extends or implements them. Furthermore, if a method is overridden in the child class or implemented from an interface, it must also fulfill the contract that has been defined for that method in the parent class or interface. We do not require an interface for every class.

Method Calls. Methods can be called either within the same class, by an object instantiating that class, or directly by that class if the method is static. The implementation of a method can be either in CorC or in Java. To verify a method call, CorC supports inlining and contracting (Knüppel et al., 2018) (i.e., inserting its implementation or using its contract as defined in the CbC method call refinement rule (see Definition 2.2). If contracting is used, it is assumed that the contract of the called method holds, but this is not explicitly checked in the method call refinement rule. As explained above, overridden methods still have to fulfill the contract of their parent class to guarantee the correctness of dynamic dispatch method calls on objects according to the Liskov principle.

Framing. In addition to a pre- and a postcondition, a frame is defined for each method, containing all variables whose data can be modified. This information helps the caller of a method to determine which parts of the state will not be changed by the method call (Borgida et al., 1995). For formal verification of object-oriented programs, framing is important because the caller implicitly knows which fields remain unchanged during the execution of a method (Ahrendt et al., 2016a; Weiß, 2011). Furthermore, framing is important for information hiding (Leavens and Muller, 2007) and for avoiding unwanted side effects (Leino and Nelson, 2002). In CorC, the frame of a method is automatically determined by traversing its refinement steps and collecting the variables that are on the left hand side of an assignment. However, the frame can also be defined manually by the user. When verifying a method with a frame, it is checked whether all variables that are not included in the frame still have the same value after execution as before the execution of this method.

Limitations

In the following, we discuss three limitations that we do not address with our object-oriented concepts in CorC 2.0.

Constructors. In addition to methods, classes contain constructors that are used to instantiate an object from a class. Although constructors are very important in object-oriented programming, we do not support their creation and verification in CorC. However, an initialization method that calls a default constructor can be created and verified with CorC as a workaround.

Exceptional Behavior. Method contracts are limited to a pre- and postcondition pair, including a frame. Exceptional behavior, such as expecting a particular exception to be thrown, cannot be expressed in the contracts in CorC 2.0.

Concurrency. Concurrency is a paradigm that can be found in many object-oriented programming languages. There are several verification approaches that specialize in verifying race conditions, deadlocks, or memory properties. Examples are VeriFast (Jacobs et al., 2011), VerCors (Blom and Huisman, 2014), Chalice (Leino et al., 2009), and OpenJML (Cok, 2021). In CbC as implemented in CorC 2.0, we currently do not support the construction of concurrent or multi-threaded programs. For this, a formalized concurrency model, such as ABS (Johnsen et al., 2011) is needed, and additional CbC refinement rules (e.g., for asynchronous method calls) need to be defined. We are aware that this is an important area of modern software engineering that we plan to address in future work.

3.1.2. An Integrated Development Process in CorC 2.0

In Figure 3.1, we give an overview of the project structure and the development process in CorC 2.0. We use the example of a bank account software system, which consists of two classes. The class `Account` has two methods, `update` and `undoUpdate`, to manipulate the balance of an account. The class `Transaction` provides the method `transfer` which allows money to be transferred from a source account to a target account, and the method `lock`, which locks an account so that the balance cannot be changed.

On the left side of Figure 3.1, we see the project structure of the bank account system. There are two folders named after the two classes, `Account` and `Transaction`, which contain all the files created by CorC for each class. The cbcclass and cbcmethod files are representations of the Java classes and methods in CbC format that can be viewed, edited, and verified by CorC. Each folder contains a cbcclass file, also named after the class, which contains all information related to that class (i.e., class and

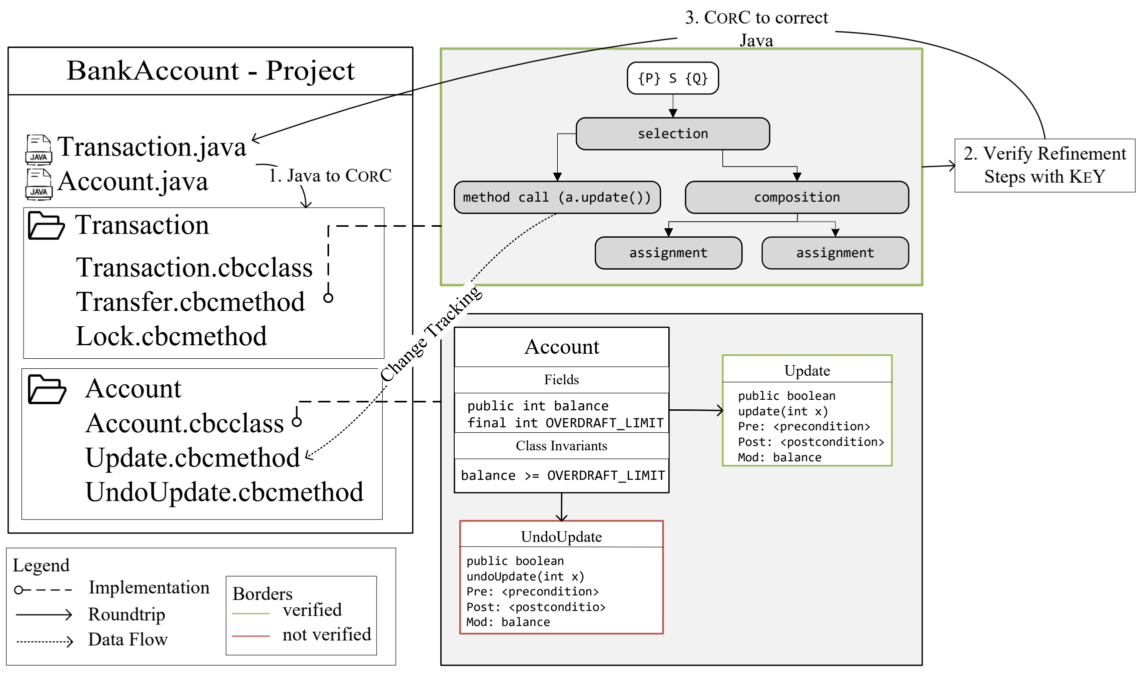


Figure 3.1.: Development process in CorC 2.0 (Bordis et al., 2022a). We show the project structure, the method editor, and the class editor. The arrows show the three steps of the roundtrip engineering process and the change tracking mechanism.

method information). The implementation of `Account.cbcclass` is shown in the bottom center of Figure 3.1, similar to a UML class diagram. There is a larger box entitled `Account` that defines the field `balance`, the constant `OVERDRAFT_LIMIT`, and a class invariant. If this class inherits from another class or implements an interface, this would be defined using the Java keywords `extends` and `implements`. The methods `update` and `undoUpdate` are shown in two separate boxes that are connected to the Account class. They show the method signature and the contract consisting of precondition, postcondition, and framing. Furthermore, their border is either green or red to indicate their verification status.

Besides the cbcclass files, there is also a cbcmethod file for each method in the class folders. The development of methods is generally the same as in CorC without object-orientation (see Section 2.2). The content of `Transfer.cbcmethod` is shown at the top center of Figure 3.1. It shows the refinement steps that are used in CorC to construct the method `transfer` starting from the starting Hoare triple `{P}S{Q}`. The precondition `P` and the postcondition `Q` are the same as the pre- and postcondition that are contained in `Transaction.cbcclass` for method `transfer`. The single refinement steps are created in CorC and verified with KeY as described in Section 2.2.2.

In the following, we describe some new core features of CorC 2.0 that are important for integrating CbC into a software development process.

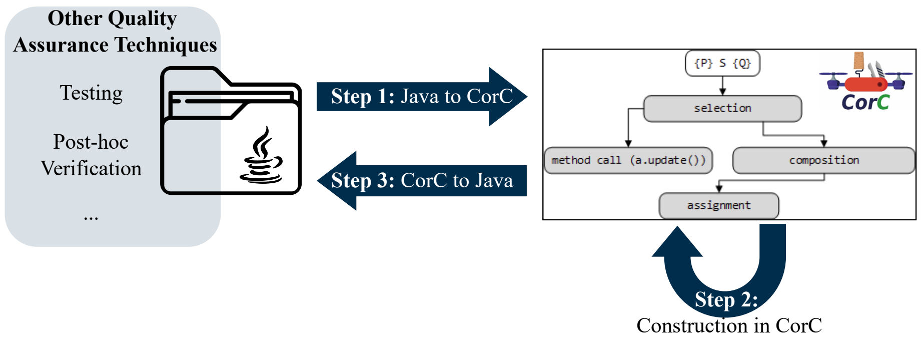


Figure 3.2.: Roundtrip engineering workflow to integrate CorC 2.0 into existing Java projects. Other quality assurance techniques, such as testing and post-hoc verification, can be applied to the Java project as usual.

Roundtrip Engineering. To seamlessly integrate CorC 2.0 into existing Java software system development processes, we introduce a *roundtrip engineering process*. This roundtrip engineering process can be used to 1) implement new methods using CbC, 2) integrate provably correct CbC implementations into existing projects, 3) better trace the source of errors when the developer fails to prove a certain method with post-hoc verification, or 4) combine CbC with other software quality assurance techniques. Either way, the correct and modified implementation and specification can be integrated back into the original project. Roundtrip engineering is performed in three steps, as shown in Figure 3.2. These steps are applied on a Java project (left) and in CorC 2.0 (right) and can be repeated iteratively to create an incremental development process.

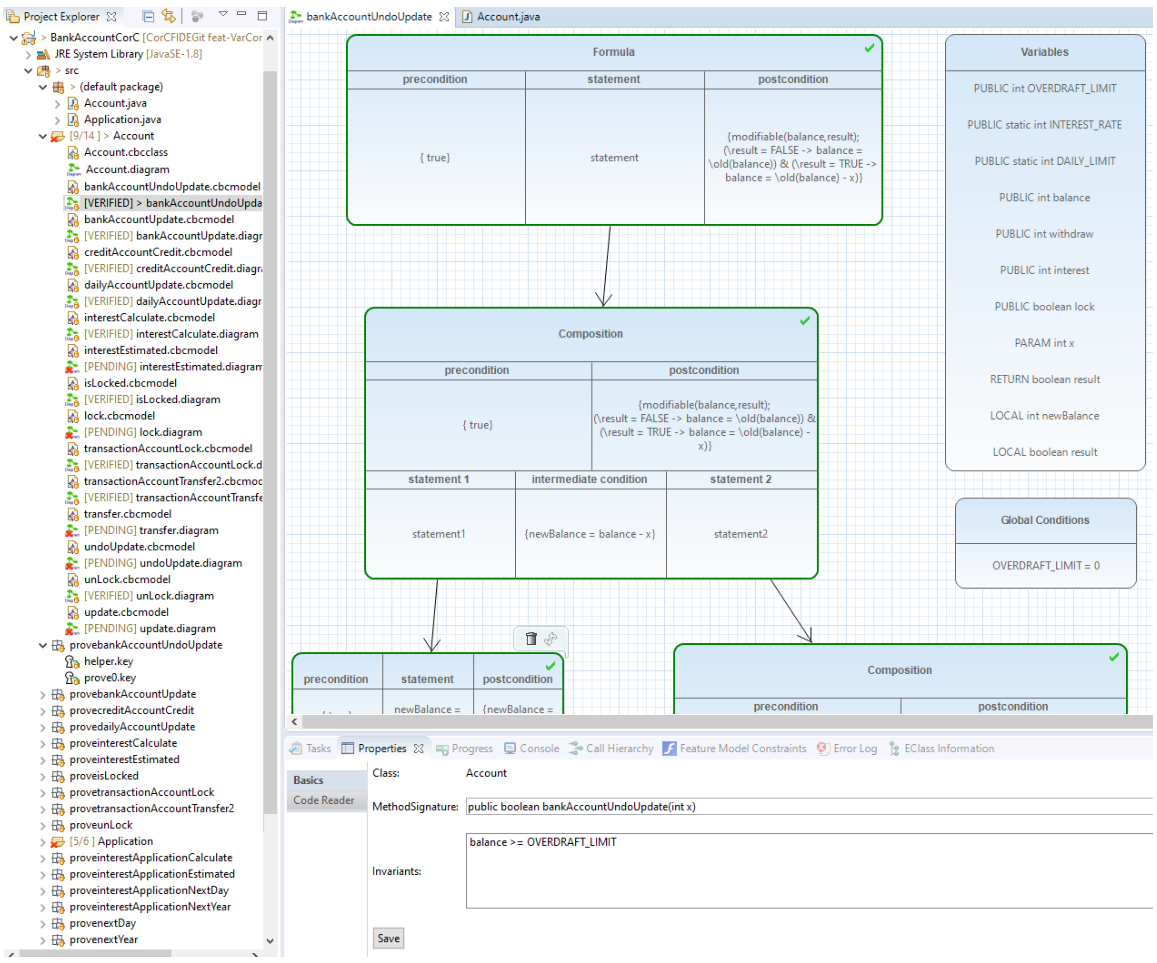
Step 1: If a project already contains Java classes that contain methods that need to be verified, the first step is to *convert* the classes and methods into cbcclasses and cbcmethods. During this step, the user selects which methods to convert and completes the missing specifications, such as intermediate conditions, loop invariants, and variants in the cbcmethods. Alternatively, the user can create cbcclasses and cbcmethods from scratch in CorC, making this first step optional. In this case, the user must apply the refinement rules manually to construct correct method implementations following CbC in the next step.

Step 2: In this step, the refinement steps in the cbcmethods are verified. Method calls can be verified using either their full implementation (inlining) or their specified contract (contracting). If contracting is used, the method can either be implemented in CorC or provided in Java with a JML contract. This allows the user to freely combine CorC with existing Java methods and other quality assurance techniques, such as testing or post-hoc verification. However, the verification of method calls with contracting does not guarantee that a called method actually fulfils its contract. With inlining, not all methods that are called need to be specified, and with contracting, methods that are specified do not necessarily need to be verified to be called and can be assumed to be correct. Consequently, a connection of verified (safety-critical) and unverified (not safety-critical) parts of a software system is possible.

Step 3: CorC can generate provably correct Java code from the verified cbcmethods, either for a new Java class or back into the original Java class from which it was imported, replacing the original implementation and contract. After this step, other quality assurance techniques, such as testing or post-hoc verification, can be applied, either to the generated code, or to other parts of the software project that have not been constructed with CbC.

Independent Step: Applying Other Quality Assurance Techniques: As mentioned at the beginning, the goal of this roundtrip engineering process is to integrate CbC into existing software development processes. This means, that we envision a combined use of CbC and other quality assurance techniques, such as software testing or post-hoc verification, to leverage the different advantages of the techniques. For example, despite verifying the functional correctness of the software in a system, it might still be necessary to perform penetration tests or integration tests, as this is something that cannot be guaranteed with CbC. In Figure 3.2, the application of other quality assurance techniques to the original Java project is shown on the left. Other quality assurance techniques can be applied in the project as usual, *before of after* the three roundtrip engineering steps have been performed, and therefore, CbC development can be flexibly incorporated into existing software development processes that use other quality assurance techniques.

Change Tracking. To further improve the usability of CorC, we introduce a notification system that keeps track of changes by setting already verified refinements to unverified. This is especially critical for methods that use method calls, inheritance, or interfaces, because their verification depends on specifications provided the called methods, the classes they are inheriting from, or the interface they are implementing. Consequently, once these specifications are changed, the verification results become invalid. For example, in Figure 3.1, the implementation of the method `transfer` calls the method `update` on `Account a`. Now, if method `update` in class `Account` is modified, the method call refinement in method `transfer` needs to be re-verified, as the old proof is based on a possibly outdated contract. The change tracking system prevents the user from overlooking these changes. It also allows direct navigation to the affected method (in our example, to method `transfer`) for re-verification. In the background, the affected refinement rules are automatically set to not verified, so that these refinement steps cannot be mistakenly assumed to be correct. Since CbC has a fine-grained structure of individual refinement steps, not all refinement steps of a method need to be re-verified, but only those that are affected by the change. CorC 2.0 can maintain the correctness of evolving software better than its predecessor CorC, since it is no longer possible to have refinements falsely marked as verified when the specifications that this proof depend on have changed.


Project Explorer
BankAccountCorC [CorCFIDEGit feat-VarCor
JRE System Library [JavaSE-1.8]
src
(default package)
Account.java
Application.java
[9/14] > Account
Account.cbcclass
Account.diagram
bankAccountUndoUpdate.cbcmodel
[VERIFIED] > bankAccountUndoUpda
bankAccountUpdate.cbcmodel
[VERIFIED] bankAccountUpdate.diagr
creditAccountCredit.cbcmodel
[VERIFIED] creditAccountCredit.diagr
dailyAccountUpdate.cbcmodel
[VERIFIED] dailyAccountUpdate.diagr
interestCalculate.cbcmodel
[VERIFIED] interestCalculate.diagram
interestEstimated.cbcmodel
[PENDING] interestEstimated.diagram
isLocked.cbcmodel
[VERIFIED] isLocked.diagram
lock.cbcmodel
[PENDING] lock.diagram
transactionAccountLock.cbcmodel
[VERIFIED] transactionAccountLock.d
transactionAccountTransfer2.cbcmoc
[VERIFIED] transactionAccountTransfe
transfer.cbcmodel
[PENDING] transfer.diagram
undoUpdate.cbcmodel
[PENDING] undoUpdate.diagram
unLock.cbcmodel
[VERIFIED] unLock.diagram
update.cbcmodel
[PENDING] update.diagram
provebankAccountUndoUpdate
helper.key
prove0.key
provebankAccountUpdate
provecreditAccountCredit
provedailyAccountUpdate
proveinterestCalculate
proveinterestEstimated
proveisLocked
provetransactionAccountLock
provetransactionAccountTransfer2
proveunLock
[5/6] Application
proveinterestApplicationCalculate
proveinterestApplicationEstimated
proveinterestApplicationNextDay
proveinterestApplicationNextYear
provenextDay
provenextYear
bankAccountUndoUpdate
Account.java
Formula
precondition
statement
postcondition
{ true}
statement
{modifiable(balance,result); (\result = FALSE -> balance = \old(balance)) & (\result = TRUE -> balance = \old(balance) - x)}
Composition
precondition
postcondition
{ true}
{modifiable(balance,result); (\result = FALSE -> balance = \old(balance)) & (\result = TRUE -> balance = \old(balance) - x)}
statement 1
intermediate condition
statement 2
statement1
{newBalance = balance - x}
statement2
precondition
statement
postcondition
newBalance =
{newBalance =
Composition
precondition
postcondition
Variables
PUBLIC int OVERDRAFT_LIMIT
PUBLIC static int INTEREST_RATE
PUBLIC static int DAILY_LIMIT
PUBLIC int balance
PUBLIC int withdraw
PUBLIC int interest
PUBLIC boolean lock
PARAM int x
RETURN boolean result
LOCAL int newBalance
LOCAL boolean result
Global Conditions
OVERDRAFT_LIMIT = 0
Tasks
Properties
Progress
Console
Call Hierarchy
Feature Model Constraints
Error Log
EClass Information
Basics
Code Reader
Save

Class: Account

MethodSignature: public boolean bankAccountUndoUpdate(int x)

Invariants: balance >= OVERDRAFT_LIMIT

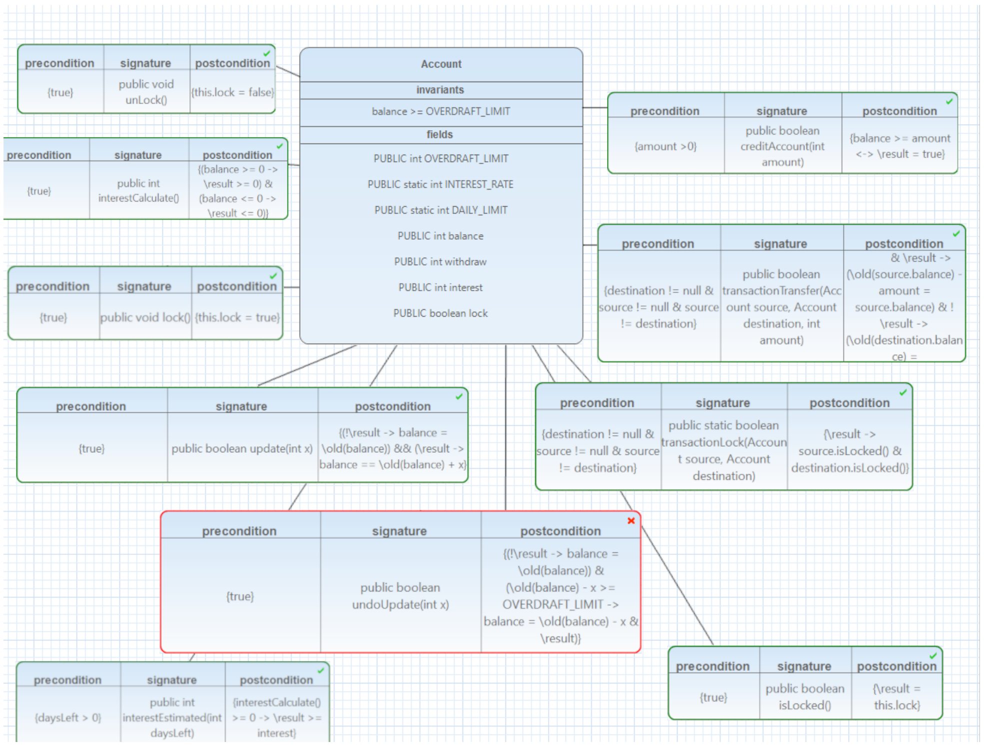
Figure 3.3.: Screenshot of CorC 2.0 with new project explorer on the left and properties view at the bottom (Bordis et al., 2022a). In the center, we show the method view with method `update`.

3.1.3. Implementation

CorC 2.0[1] is an open-source Eclipse plug-in that supports the development of object-oriented programs using CbC as described in this chapter. The basic implementation of CorC has already been described in Section 2.2.2 and remains the same. In this section, we only describe the new functionalities. In Figure 3.3, we show a screenshot of the graphical view for developing methods in CorC 2.0. The new parts of the graphical editor in this screenshot is the project view on the left and the properties view at the bottom. The functionality of the view in the center where we can see the CorC diagram of method `undoUpdate`, remains unchanged (see Section 2.2).

On the left is the project structure of the *Bank Account* subject system that we used as an example in the previous subsection. The Java classes are in the default package. Then, there are folders named after the classes, containing information about the

[1] https://github.com/KIT-TVA/CorC



precondition	signature	postcondition
{true}	public void unLock()	{this.lock = false}

Account

invariants

balance >= OVERDRAFT_LIMIT

fields

PUBLIC int OVERDRAFT_LIMIT
PUBLIC static int INTEREST_RATE
PUBLIC static int DAILY_LIMIT
PUBLIC int balance
PUBLIC int withdraw
PUBLIC int interest
PUBLIC boolean lock

precondition	signature	postcondition
{amount >0}	public boolean creditAccount(int amount)	{balance >= amount <-> \result = true}

precondition	signature	postcondition
{true}	public int interestCalculate()	{(balance >= 0 -> \result >= 0) & (balance <= 0 -> \result <= 0)}

precondition	signature	postcondition
{destination != null & source != null & source != destination}	public boolean transactionTransfer(Account source, Account destination, int amount)	& \result -> (\old(source.balance) - amount = source.balance) & ! \result -> (\old(destination.balance) =

precondition	signature	postcondition
{true}	public void lock()	{this.lock = true}

precondition	signature	postcondition
{true}	public boolean update(int x)	{(!\result -> balance = \old(balance)) && (\result -> balance == \old(balance) + x}

precondition	signature	postcondition
{destination != null & source != null & source != destination}	public static boolean transactionLock(Account source, Account destination)	{\result -> source.isLocked() & destination.isLocked()}

precondition	signature	postcondition
{true}	public boolean undoUpdate(int x)	{(!\result -> balance = \old(balance)) & (\old(balance) - x >= OVERDRAFT_LIMIT -> balance = \old(balance) - x & \result)}

precondition	signature	postcondition
{daysLeft > 0}	public int interestEstimated(int daysLeft)	{interestCalculate() >= 0 -> \result >= interest}

precondition	signature	postcondition
{true}	public boolean isLocked()	{\result = this.lock}

Figure 3.4.: Screenshot of the class editor showing class `Account` in CorC 2.0 (Bordis et al., 2022a).

class and all its methods in form of diagram and model files named after the individual classes and methods. The diagram files (⟨*methodName*/*className*⟩.*diagram*) contain the graphical representation, and the model files (⟨*methodName*⟩.*cbcmodel*/ ⟨*className*⟩.*cbcclass*) store the information about the methods and classes in the corresponding meta-model. The *prove*⟨*methodName*⟩ folder stores generated proof files, which contain the side conditions for a refinement step that need to be verified. Each proof file is (semi-)automatically verified using KeY. For a better overview, the folder name is preceded by the total number of verified methods and the ⟨*methodName*⟩.*diagram* files are preceded by the verification status. In this context, *verified* means that all refinement steps of a method could be proven and *pending* means that at least one refinement step has not yet been successfully verified.

At the bottom of Figure 3.3, we can see the properties view with the currently active *Basics* tab. It shows more information about the method `undoUpdate`, such as the class it belongs to, its signature, and the class invariants it must fulfill. The method signature can also be edited. To change any other information, the class file must be opened. The other tab in the properties view is called *Code Reader* and displays the selected Java statements or specifications in the diagram in a larger window with syntax highlighting. This is useful for making particularly long specifications easier to read and modify without introducing syntax errors.

In Figure 3.4, we show a screenshot of the class view in CorC 2.0. It displays the content of file *Account.diagram*. At the top center, we see a class definition component that looks similar to a UML class diagram. At the top, it displays the name of the class, and below that, it lists the class invariants and fields with their visibility, type, and name. Surrounding the class component, there are several other components, which are the methods that are implemented in that class. For each method, its signature and contract is shown. Additionally, the verification status of a method is indicated by red and green borders. In this view, new methods, fields, and class invariants can be added, and existing ones can be edited. Changed information, e.g., a changed precondition of a method or a changed type of a field, is directly available in all method diagrams, as each method diagram directly refers to its cbcclass file. In case of a changed precondition, the user is notified by the change tracking mechanism, and the verification status is set to not verified, as described in the previous subsection. For an easy navigation to the method diagrams, the user can double-click on a method component.

3.2. Evaluation

In this section, we evaluate CorC 2.0 by comparing it to post-hoc verification. We use KeY for post-hoc verification, because KeY is the built-in verifier in CorC, but KeY can be understood as a synonym for post-hoc verification. To evaluate whether it is feasible to construct correct programs with CorC, we want to evaluate CorC 2.0 quantitatively and qualitatively. Specifically, we want to answer the following two research questions:

RQ1 (proof efficiency): How do the verification time and proof nodes of verifying programs in CorC compare to post-hoc verification in KeY?

RQ2 (usability): How does the CorC development process assist in creating correct programs in comparison to the assistance of KeY for post-hoc verification?

The first research question is answered by creating three subject systems using both CorC and KeY, and measuring the verification time and the number of proof nodes. Each subject system consists of several Java classes with specified methods. Each method is constructed and verified with CorC. For post-hoc verification, we verified the methods written in Java and specified with JML (precondition, postcondition, and loop invariants, but no further intermediate specifications were given). For method calls, we used contracting to prove correctness. We used KeY as an automatic verification tool. We measured the number of proof nodes by the number of rule applications of KeY. This metric gives insights into the proof complexity, the more proof nodes KeY needs for a proof, the more complex it was. The number of proof nodes remains consistent among multiple verification runs. The verification time is the time that KeY needs for the proof. It changes for each verification run and therefore, we performed the verification task five times and calculated the averages. We do not consider the manual effort of writing additional specifications for CorC. For the second research question, we discuss CorC qualitatively by referring to two

Subject System	#Classes	#Methods	#Verified with CorC	#Verified with KeY
BankAccount (Thüm et al., 2012)	2	10	10	9
Email (Hall, 2005)	2	12	12	8
Elevator (Plath and Ryan, 2001)	5	18	18	17

Table 3.1.: Characteristics of the subject systems (Bordis et al., 2022a).

user studies (Runge et al., 2020; Runge et al., 2021) and report on our experiences of using object-orientation and the roundtrip engineering process in CorC 2.0.

Subject Systems. We implemented and verified three subject systems using CorC and KeY. We chose the subject systems *BankAccount* (Thüm et al., 2012), *Email* (Hall, 2005), and *Elevator* (Plath and Ryan, 2001) because they are implemented in an object-oriented way, such that we can evaluate the new feature of CorC 2.0. In Table 3.1, we show some characteristics of the subject systems. The *BankAccount* subject system has been briefly introduced in Section 3.1.2. It implements basic functionality of a bank account, such as updating its balance and transferring money from one account to another. It consists of two classes and 10 methods. The *Email* subject system is implemented in two classes and 12 methods, providing the basic functionality of an email client that can receive, send, and forward emails. The *Elevator* subject system implements a passenger elevator that can move up and down. Persons can call the elevator on each floor and enter and leave the elevator. The *Elevator* subject system is implemented in five classes and 18 methods. The size of the methods across all subject systems ranges from 1 to 20 lines of code.

3.2.1. RQ1 – Verification Time and Proof Nodes

To answer the first research question, we constructed all 40 methods in CorC following CbC. For post-hoc verification, we used the same pre-/postcondition specifications as in CorC, but KeY failed to prove six algorithms automatically (see Table 3.1). For example, the verification of methods with KeY failed in the steps where the loop must be proven. In CorC, the verification of loops is split into several smaller proofs, which reduces the overall proof complexity. Another reason for the reduced proof complexity in CorC is that intermediate specifications split the verification task into smaller proofs. In contrast, using post-hoc verification is more coarse-grained, since only preconditions, postconditions, and loop invariants are specified. We observed that debugging verification problems that cannot be verified automatically with post-hoc verification is more challenging than debugging the same problem constructed with CbC in CorC.

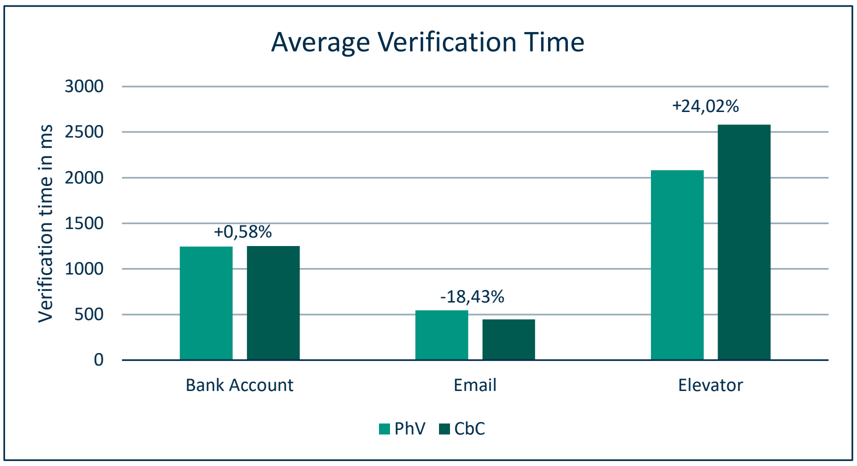


Figure 3.5.: Average verification time of the subject systems with CbC and post-hoc verification (PhV) (Bordis et al., 2022a).

In Figure 3.5, we show the average verification time measured in milliseconds, and in Figure 3.6, we show the proof nodes for each subject system, but only for the 34 methods that could be verified with both, CorC 2.0 and KeY. The verification time ranges from 0.1 seconds for some methods to 16 seconds for the most complex methods. For 22 methods, the verification time with CorC 2.0 is faster, for 12 methods the verification with KeY is faster. The biggest differences are: `enterElevator` is 268% faster with CorC 2.0, `addWaitingPerson` is 271% faster with KeY. The average verification time for the *Email* subject system shows that the verification is 18.43% faster with CorC 2.0, but the *Elevator* subject system is on average 24.02% slower with CorC 2.0. For the number of proof nodes, we got similar results. In 26 cases, CorC 2.0 needed less proof nodes, and in 8 cases, KeY needed less proof nodes. For the subject systems *BankAccount* and *Email*, CorC 2.0 needed 24.1% and 23,78% less proof nodes, and for the *Elevator* CorC 2.0 needed 29.29% more nodes. Overall, the results are in the same order of magnitude. The exact verification times and number of proof nodes for each method are shown in Appendix A.

Discussion. There is no trend identifiable in the proof efficiency in terms of verification time and proof nodes. Therefore, we cannot answer the research question that the verification with CbC is more efficient than with post-hoc verification in terms of verification time or number of proof nodes in the affirmative. The verification time and the number of proof nodes are similar for both approaches. However, we were able to verify more methods in total with CorC. The additional specifications in the form of intermediate annotations and the splitting into several smaller proofs facilitate the easier completion of proofs, but do not significantly affect the verification effort. These results are in contrast to previous evaluation results (Runge et al., 2019), where a reduced proof efficiency for CbC was found when verifying single algorithms. Since we are promoting the use of CbC and post-hoc verification in concert, a similar

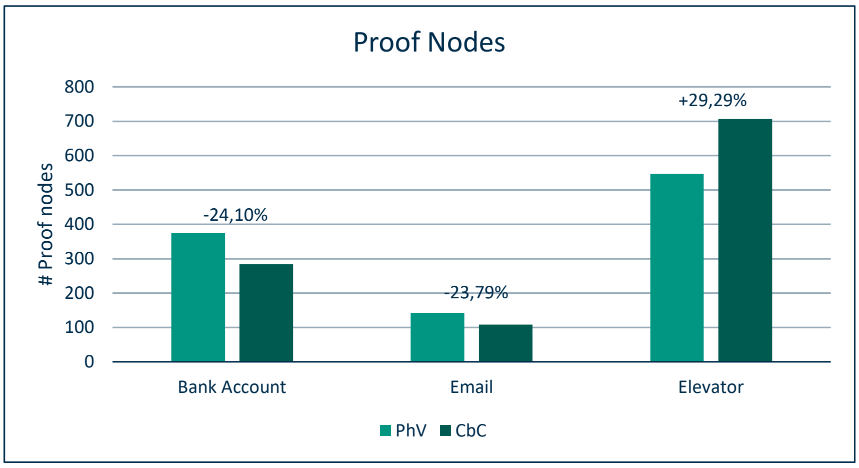


Figure 3.6.: Proof nodes of the subject systems with CbC and post-hoc verification (PhV) (Bordis et al., 2022a).

verification effort is beneficial. There is no discernible disadvantage in terms of proof efficiency in using either approach.

3.2.2. RQ2 – Usability of CorC 2.0

For the second research question, we first summarize the results of two user studies (Runge et al., 2020; Runge et al., 2021) which compare the usability of CbC and post-hoc verification. Both user studies have been conducted using CorC without object-orientation and the roundtrip engineering process, as proposed in this chapter. We still summarize these results because the CbC approach to develop single methods is the same and remains the most important feature of CorC. Afterwards, we give an experience report for creating the *BankAccount*, *Email*, and *Elevator* subject systems using the new features of CorC 2.0, which are object-orientation, the roundtrip engineering process, and the change tracking mechanism.

Summary of Two User Studies. For the second research question, we first summarize the results of two user studies (Runge et al., 2020; Runge et al., 2021) with a total of 23 students. In both studies, a group of students was divided in half. One group created and verified a method with CorC and a second method with KeY. The second group created the same methods, but used the tools in reverse order. We analyzed the faults in the developed and verified methods and conducted a usability questionnaire on CorC and KeY at the end of the user study. The first user study was conducted in person in 2019 (Runge et al., 2020), the second user study was conducted online in 2021 (Runge et al., 2021). In both user studies and with both tools, we had several correct implementations that were not verified in the given time frame. A common problem was that the loop invariant was too weak. In the usability questionnaire, most participants preferred CorC to KeY because of better and more fine-grained feedback when errors occurred during the verification. It was easier to detect and solve faults with CorC. A minority of participants preferred KeY because they were more familiar with the syntax of Java and JML. Since we now support roundtrip development with CorC 2.0, we believe that this statement is weakened. Users can now freely develop and verify programs with CorC or KeY and automatically generate the program for the other approach. Thus, the preferred tool can be used without restrictions.

Experience Report. The three subject systems that we used to evaluate the first research question **RQ1** already had Java implementations and JML specifications. To transfer these into CorC 2.0, we used the first step of CorC 2.0's roundtrip engineering process, which creates cbcclass and cbcmethod files. All fields, class invariants, and method signatures were correctly transferred into cbcclass files. The implementations of the methods were also correctly translated into corresponding cbc refinement trees in the cbcmethod files. Without the automation of the roundtrip engineering process, this would have been approximately one working day of manual effort. For the methods, we additionally provided intermediate conditions in the CbC refinement trees. In some cases, we had to slightly adjust the refinement trees and add another assignment refinement. For example, we noticed that Java for-loops were not correctly translated into the repetition refinement rule, which works similar to a while-loop. Therefore, we had to initialize a local variable before the repetition refinement rule, which was originally part of the Java for-loop. In the second step of the roundtrip engineering process, we verified that the side conditions of the applied refinement steps were correct. We modified the intermediate conditions as needed until all refinement steps in all methods could be proven. During this step, we found CorC 2.0's change tracking feature especially valuable. When we verified a refinement step that called another method, we sometimes had to change the contract of the called method. The change tracking feature then marked all affected method calls in all CorC diagrams as "not verified". With this feature, we did not miss any open verification obligations. In summary, we can confirm that CorC 2.0 helps to develop correct software. Additionally, the new features of CorC 2.0 support the implementation of object-oriented code.

3.2.3. Threats to Validity

In this section, we discuss threats to validity for the quantitative and qualitative evaluation of object-oriented CbC as implemented in CorC 2.0. In the quantitative evaluation, we compared the verification time and number of proof nodes with post-hoc verification as implemented in KeY. The qualitative evaluation consisted of two parts. First, we summarized the results of two user studies (Runge et al., 2020; Runge et al., 2021) that compared CbC development with post-hoc verification on method level. Second, we reported on our experience of using CorC 2.0 for the construction of the subject systems. Threats to validity for the user studies have already been discussed in detail by Runge et al. (2020) and Runge et al. (2021). Therefore, we only summarize these threats to validity and focus on the experience report.

RQ1 – Quantitative Evaluation

External Validity. Our main concern with the quantitative evaluation is generalizability. The methods implemented in the three subject systems range in size from 1 to 30 lines of code. These small methods reduce the generalizability to larger algorithmic problems, and general conclusions about the verification effort between post-hoc verification and CbC. However, the size of the subject systems give us the opportunity to analyze each subject system in detail. Although CorC 2.0 extends the scope to object-orientation, we still consider CorC to be a tool for smaller, but challenging algorithmic problems. Nonetheless, it is important that algorithms constructed with CbC can be integrated into bigger object-oriented software systems and that they fulfill the properties of these systems. Due to the roundtrip engineering process, CorC 2.0 can now be combined with other tools to scale the size of systems to which CbC and CorC can be applied. Another assumption that we made was that the verification of the subject systems had to work automatically. In general, there are few open-source projects in the literature that meet this criterion.

Internal Validity. We wrote most of the specifications and implementations for the three subject systems ourselves. Thus, there may still be faults in the specifications or implementations. Since we were able to verify all of the methods with CorC and most of the methods with KeY, this is a strong indication of the correctness of the subject systems. In particular, we checked the equality of the specifications for the two different approaches.

RQ2 – Qualitative Evaluation

External Validity. The design of the qualitative user studies lacks generalizability, as only 23 participants in total took part. However, this enabled a detailed observation of each change that a participant made during the experiment. Runge et al. (2020) and Runge et al. (2021) value the individual insights gained from these observations over a generalizable user study with many participants. Additionally, the results of the user studies are not generalizable to larger software projects because only small algorithmic problems were used. This was necessary, because on the one hand, the participants are non-experts in software verification, and, therefore, the experiments had to be simple enough for them to solve. On the other hand, the experiments needed to be complex enough to give the participants practical experience in using techniques implemented in CorC and KeY so they could report their opinions. The algorithmic problems that were used fulfilled these criteria, and helped us in this evaluation to qualitatively assess the difference in developing methods with CbC compared to post-hoc verification, because the evaluated part of the implementation was unchanged in CorC as used in the user studies and CorC 2.0 as shown in this chapter. Still, some features of CorC 2.0 were not evaluated by the participants of the user studies (e.g., class view, roundtrip engineering process, and change tracking mechanism). Instead, they were discussed based on our personal and internal usage in the experience report. This limits the generalizability of those findings, because usability was not tested in experiments with external participants but rather inferred from developer perspectives, which may not fully represent user experience. Therefore, conclusions drawn from this qualitative evaluation, especially those regarding features that were only evaluated in the experience report, should be considered preliminary rather than definitive.

Internal Validity. The group of participants in the user studies was homogeneous. They were all computer science students who took a course on post-hoc verification and CbC development. Therefore, they were classified as junior software engineers with the necessary background in code understanding, as well as the necessary theoretical and practical background required to use CorC and KeY. While their feedback may not fully represent challenges faced by software engineers with different backgrounds, they all shared a basic understanding of the evaluated approaches (CbC and post-hoc verification) required to conduct this kind of user study. Another threat to the experimental design of the user studies was the limited time frame. Participants only had 30 minutes to implement each method. With more time, more participants might have been able to verify the assigned methods and complete the loop invariants.

In the experience report, we discussed the usability of CorC 2.0's new features, which we developed ourselves. This may introduce confirmation bias when interpreting usability issues. Since we did not document any other metrics besides our observations, the interpretation of some findings in the experience report may also be incomplete. Despite these limitations, our evaluation provides valuable insights into the feasibility and usability of object-oriented CbC. We assessed single method development in the user studies and while using CorC 2.0 ourselves, we were able to save time on implementation tasks and locate and fix bugs in the implementation. In the future, CorC 2.0 serves as a foundation for further large-scale empirical studies.

3.3. Related Work

In this section, we position our object model and roundtrip engineering process in the context of related work on 1) CbC, 2) other refinement-based software construction approaches, and 3) post-hoc verification.

Scaling CbC. Knüppel et al. (2020b) propose the ArchiCorC framework for scaling CbC to specify and reason about component architectures. In ArchiCorC, a user models UML-style component diagrams with required and provided interfaces, where methods contained in interfaces are associated with contracts and mapped to CbC implementations. In contrast to the object-oriented programming model that we introduced, ArchiCorC only supports interfaces and methods, but no sub-classing. The focus of ArchiCorC is on the connection between component interfaces and CbC method implementations. The workflow of ArchiCorC also includes the generation of correct Java code, which is similar to the third step of our roundtrip engineering process. However, ArchiCorC's workflow does not include steps to import Java code from existing projects.

Object-oriented Programming in Refinement-based Software Construction Approaches. Besides CbC, as we pursue in this work, there are other refinement-based approaches that guarantee the correctness of a program under development. We already discussed their relation to CbC in Section 1.1. In the following, we only briefly mention differences and focus on their integration of object-orientation and an integrated development process, if there is any. Event-B is a framework for refinement-based development of system-level automata based on events that are defined with a set of conditions and actions that are performed if all conditions of an event are fulfilled. Méry and Singh (2011) present a translation tool that translates these `Event-B` models into a target programming language following a multi-phased process, including a syntax-directed translation, event scheduling for optimization, and verification of the generated code. The generated code consists of a set of functions (resembling the set of events in the `Event-B` model) with selections that use the event conditions in their guards and execute the actions in the form of assignments. Since the abstraction level of `Event-B` system-level models is different from CbC, the generated code is also different. CbC's refinement rules directly correlate to programming constructs, and therefore, CbC refinement trees can directly be translated into code, similar to an abstract syntax tree. Even though the abstraction level of CbC and `Event-B` is different, the generated code could possibly be combined. For example, CorC 2.0 could be used to construct a more complex action that is executed by an event from the `Event-B` model. Neither ArcAngel (Oliveira et al., 2003; Zeyda et al., 2009) nor SOCOS (Back et al., 2007) support object-orientation and a process similar to our roundtrip engineering process.

Post-hoc Verification for Object-oriented Software. There are many verification tools that support object-oriented programming languages. We have already mentioned KeY (Ahrendt et al., 2016a) for verifying Java programs with JML (Leavens and Cheon, 2006) specifications. More specifically, KeY supports classes, inheritance, dynamic dispatch, and polymorphism using class invariants combined with the Liskov principle for inheritance and interfaces as well. Similar to KeY, ESC/Java2 (Chalin et al., 2006) and OpenJML (Cok, 2021) provide automated reasoning for object-oriented Java programs annotated with JML specifications. The verifiers differ in the underlying technique that is used to prove the properties. OpenJML translates JML specifications into logical constraints for an SMT solver. Alternatively, OpenJML can be used for runtime assertion checking. ESC/JAVA2 performs an extended static analysis using heuristics and, hence, is neither sound nor complete, and can lead to false-positives. When introducing our object-oriented programming model to CORC 2.0, we based it on the object-oriented programming models of these three post-hoc verification tools. Therefore, we introduced specification constructs for object orientation, such as class invariants, to CORC 2.0 that were already of the concepts in other post-hoc verification tools. Some specification constructs, such as exceptional behavior of methods or ghost fields, are supported by the mentioned post-hoc verification tools but are not yet supported by CORC 2.0 and are planned for future work. Since the mentioned post-hoc verification tools work on actual JAVA code, there is no need for an additional integration into existing software development processes, like we proposed for CORC 2.0 with the roundtrip engineering process.

AUTOPROOF (Furia et al., 2017) is an automated verifier for EIFFEL, a real object-oriented programming language (meaning that everything is an object) that incorporates design-by-contract into the code. AUTOPROOF supports functional correctness verification, like CORC 2.0, but is tailored to EIFFEL instead of JAVA. KRAKATOA (Filliâtre and Marché, 2007) is also an automated verifier, but for JAVA programs that are specified with a variant of JML. KRAKATOA supports basic object-oriented constructs, such as classes and methods. However, it does not support reasoning about class invariants, inheritance, polymorphism, or framing because its focus lies more on verifying algorithms. SPEC# (Barnett et al., 2005) focuses on complex problems of object-orientation (e.g., object hierarchies and collaborative patterns) for an annotation-based dialect of C#. It has an ownership model that helps to verify properties, such as encapsulation and safe object mutation in the presence of aliasing. VERIFAST (Jacobs et al., 2011) is based on separation logic and has been designed to verify memory safety and functional correctness of object-oriented JAVA programs with a focus on complex, heap-manipulating data structures. While CORC 2.0 can reason about key features of object-orientation, such as inheritance, polymorphism, and dynamic dispatch, more complex problems, such as object hierarchies and memory safety, especially in combination with heap-manipulating data structures, have not yet been considered.

DAFNY (Leino, 2010) is a verification-aware programming language that combines object-orientation, functional programming, and imperative concepts. It does not support method overriding, like most object-oriented languages do. Another difference between DAFNY and CORC is the interpretation of the method frame. In DAFNY, it is defined that it is not allowed to *write* to any memory location in the frame, while in CORC it is defined that the value of the memory locations may not be modified. DAFNY includes several compilers that can compile proven DAFNY code into different programming languages, such as JAVA, C#, or PYTHON. Thus, despite DAFNY using its own language, it can be integrated into projects that are based on one of the compilable languages. The idea behind this is very similar to the roundtrip engineering process that we propose for CORC.

3.4. Chapter Summary

The goal of this chapter was to investigate, how CbC, originally designed for single algorithms, can be extended to object-oriented software engineering, incorporating modern code structures, and integrating a software development process. In particular, key questions were how to integrate and guarantee the correctness in an object-oriented programming model, and how to integrate CbC into existing software development processes, such that CbC can be used in combination with other quality assurance techniques. We extended CbC with core object-oriented features: classes with invariants, fields, methods, interfaces, and framing for method calls enabling encapsulation, inheritance, polymorphism, and dynamic dispatch. Furthermore, we proposed a three-step roundtrip engineering process that allows CorC 2.0 to be integrated with any quality assurance technique that can be applied to Java code (e.g., unit testing or post-hoc verification with KeY). By introducing object-orientation into CbC and enabling its integration with existing software development processes, we extend its applicability beyond algorithm verification, making it a practical option to be applied in modern software engineering. In addition to these conceptual extensions, we presented the implementation of CorC 2.0, the successor to CorC, which adds a graphical interfaces for object-oriented development and usability features, such as a properties view and a change tracking mechanism that prevents incorrect proof assumptions caused by changed specifications. In the evaluation, we compared object-oriented CbC (in CorC 2.0) with post-hoc verification (in KeY) in terms of proof efficiency and usability. The evaluation results do not indicate that one of the approaches, CbC or post-hoc verification, is superior in terms of proof efficiency and usability. Instead, we conclude that a combined use can leverage the individual benefits of both methodologies. In fact, this underlines our hypothesis that the combined use of different quality assurance techniques is a good strategy to develop correct software. The introduction of objects to CbC and the implementation of a roundtrip engineering process are crucial to achieving this goal for CbC and CorC 2.0.

4. Improving Correctness-by-Construction Engineering with Increasing Levels of Correctness Guarantees

The content of this chapter is based on the paper "Improving Correctness-by-Construction Engineering by Increasing Levels of Correctness Guarantees" (Bordis et al., 2025 (Submitted)), which is currently under review.

The amount of software in safety-critical systems is steadily increasing, which requires strong guarantees of the functional correctness of a program (Hoare, 2003). In practice, software testing is state-of-the-art, although formal methods, such as post-hoc verification or CbC, can provide stronger correctness guarantees. We believe that there is a usability gap between formal verification tools and software testing, which leads to formal methods not being used despite the safety-criticality of a system. Furthermore, expertise in this area is scarce, as formal methods are rarely taught at universities or during apprenticeships.

We believe that formal methods should not remain a field for experts, but that correct software must be realizable by developers with any background. We also believe that CbC has the right characteristics to be a good entry point for creating correct software, because thinking about the problem to be solved is decoupled and done before it is implemented in a program. To investigate these claims, we conducted two user studies with students who had just learned the basics of formal methods to compare CbC with CorC and classical post-hoc verification with the deductive verification tool KeY (Runge et al., 2020; Runge et al., 2021). We found that the students generally preferred the finer-grained, refinement-based approach of CbC. However, they struggled to fully implement and verify their tasks with both approaches, CbC in CorC and post-hoc verification in KeY. This leads us to believe that developing programs and proving their correctness should be further supported for developers without a strong background in formal methods (Challenge 3).

For CbC, we therefore envision a development process with three increasing levels of correctness guarantees to make it faster and easier for non-experts in formal methods to develop correct programs. Our goal is to integrate methodologies from standard software engineering that are already familiar to software developers to make verification tasks more accessible. We introduce the following three levels of correctness:

Level 1: Specified. Developers reason about the program by formally defining the expected behavior of the program before implementation. Although this correctness level does not provide a concrete correctness guarantee, it does initiate a thought process about the program as a whole, rather than starting with the implementation immediately. This correctness level is the basis for the following two correctness levels.

Level 2: Tested. Test cases are automatically generated from the provided specification and executed to detect major errors in the control flow of the developed program. Testing is common practice in software engineering, and errors are reported with concrete input/output values.

Level 3: Verified. This level of correctness provides the strongest correctness guarantee because it detects fine-grained errors and corner cases. Since most defects should already have been found at the previous levels of correctness, formal verification should be easier and faster than without the previous correctness level *tested*.

In this chapter, we integrate these three levels of correctness into CbC and the tool CorC to provide developers with understandable feedback. By addressing coarse-grained and control flow-related defects before the formal verification step, we streamline the development process, making it faster and easier. With this chapter, we address Challenge 3 (high entry barrier) and answer Research Question 2: *How can we lower the entry barrier for non-experts in by-Construction development?* Our approach addresses a common drawback of traditional verification tools, which often provide error reports without a clear explanation of their origin. By implementing multiple levels of correctness guarantees, we bridge this gap and provide developers with comprehensive feedback.

To support developers at each correctness level, we include the proposed techniques and tools in CorC. At the correctness level *specified*, we integrate exception handling with error messages that include possible reasons for exceptions. For correctness level *tested*, we provide test case generation from the specification, together with test reports after test execution. Preconditions serve as test input, and postconditions serve as test oracles. Besides the traditional test case generation for complete methods, we also support test case generation for single refinement steps to support a refinement-based CbC development style. Finally, at the correctness level *verified*, we use KeY for deductive verification, supported by counterexample generation. Counterexamples allow developers to explore concrete cases that do not meet the specification when a proof fails.

To evaluate our three-level development process, we conduct a quantitative mutation-based analysis and a qualitative usability refinement study. With the quantitative mutation-based analysis, we want to determine how effectively the newly introduced correctness level *tested* detects errors and, therefore, to evaluate the effectiveness of our three-level process as a whole. The qualitative usability refinement helps us to further improve the usability of our tool CorC by collecting feedback and suggestions for improvement for the tool support at each correctness level.

This chapter is organized as follows: In Section 4.1, we give an overview of the three-level development process by developing an example. In Sections 4.2 to 4.4, we describe how each level of correctness is realized conceptually and implemented in the tool CorC. In Section 4.5, we evaluate the three-level development process quantitatively with a mutation-based analysis and qualitatively with a usability refinement study. Finally, in Section 4.6, we position our three-level process in related work and conclude this chapter in Section 4.7.

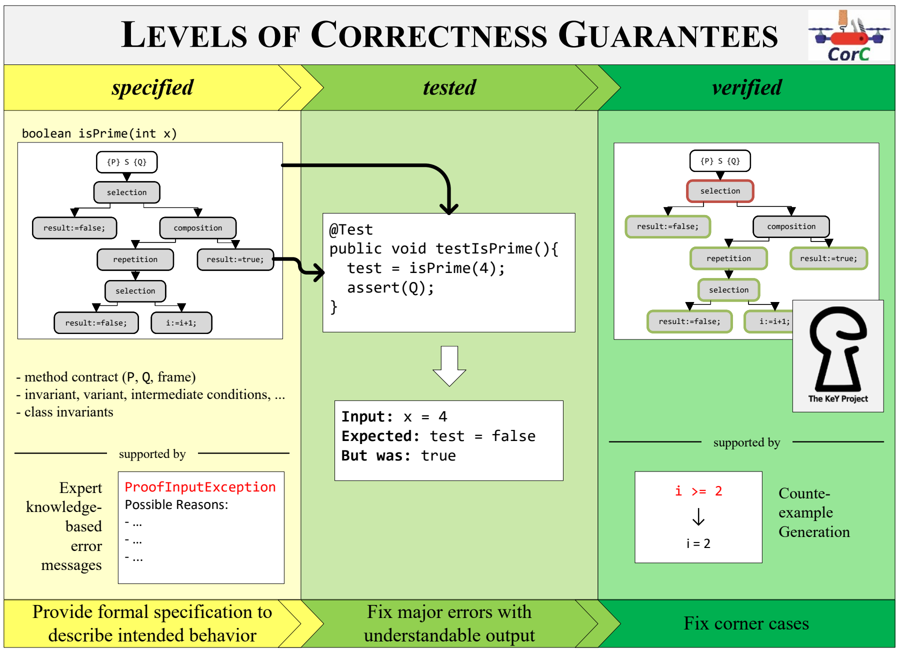
4.1. Three-Level Development Process

In Section 2.2, we explained the CbC development process and our tool CorC, which supports CbC-based program development. In two user studies with CorC (Runge et al., 2020; Runge et al., 2021), we found that there is an entry threshold for formal verification of software, not only in CorC, but also in deductive verification tools, such as KeY (Ahrendt et al., 2016a). Inspired by these results, we propose a three-level development process with the goal of lowering this entry barrier for software developers who are not experts in the field of software verification. In this section, we give an overview of our three-level development process by walking through the construction of an exemplary method `isPrime(int x)`, which should return `true` if parameter `x` is a prime number and `false` otherwise. The correctness levels are described sequentially, and for each correctness level, we explain the development steps of our method, how we support the developer in our tool CorC to reach that correctness level, the types of errors that can be found, and the resulting correctness guarantee. In Figure 4.1, we give a graphical overview of the whole development process to follow along with our explanations.

Level *Specified*. As described in Section 2.2, CbC is a specification-first approach that encourages the developer to first define a formal specification that describes the desired functionality before writing the code. After the initial specification is defined, refinement rules are applied to incrementally create code that satisfies the specification. For some refinement rules, the developer may need to add additional specifications, such as loop invariants or intermediate conditions.

Example 4.1. *We begin constructing method* `isPrime(int x)` *by defining a specification in the form of a precondition* `P` *and a postcondition* `Q`*:*

```
P: true
Q: (x ≥ 2 ∧ ¬hasFac(x) → result = true) ∧ (x < 2 ∨ hasFac(x) → result = false)
```


LEVELS OF CORRECTNESS GUARANTEES
CorC
specified
tested
verified
{P} S {Q}
selection
result:=false;
composition
repetition
result:=true;
selection
result:=false;
i:=i+1;
- method contract (P, Q, frame)
- invariant, variant, intermediate conditions, ...
- class invariants
supported by
Expert knowledge-based error messages
ProofInputException
Possible Reasons:
- ...
- ...
- ...
The KeY Project
supported by
Counte-example Generation
Provide formal specification to describe intended behavior
Fix major errors with understandable output
Fix corner cases

```
boolean isPrime(int x)
```

```
@Test
public void testIsPrime(){
  test = isPrime(4);
  assert(Q);
}
```

```
Input: x = 4
Expected: test = false
But was: true
```

```
i >= 2
↓
i = 2
```

Figure 4.1.: Process overview with three successive levels of correctness guarantees: *specified*, *tested*, and *verified* (Bordis et al., 2025 (Submitted)). The correctness levels are processed from left to right with increasing guarantee of correctness.

The precondition `P` *is the weakest precondition (i.e.,* `true`*), which means that method* `isPrime` *guarantees its postcondition* `Q` *for any state in which it is called. The postcondition* `Q` *formally defines the behavior of the method, which should return* `true` *if and only if* `x` *is a prime (i.e., if it is greater than or equal to 2 and has no factorization other than 1 and itself) and* `false` *otherwise. Therefore, it uses the predicate* `hasFac(int y)`*, which returns* `true` *if and only if* `y` *has a factorization other than 1 and* `y`*, and* `false` *otherwise.*

The pre- and postcondition then form the starting Hoare triple `{P} S {Q}` *with the abstract statement* `S`*. This triple can now be refined by applying the refinement rules from Definition 2.2. For example, we can first apply the selection refinement rule to create a case distinction between numbers greater than or equal to 2 and numbers less than 2. The refinement steps of the finished method are shown in Figure 4.2.*

In the original CbC approach, each time a refinement rule is applied, the correctness of that refinement step is guaranteed by verifying the side condition defined in the refinement rule. However, for our three-level process, we only verify those later at the third correctness level *verified*. Instead, at the correctness level *specified*, we first focus on creating a specification that describes the behavior of the program, constructing the program in form of CbC refinement steps, and checking only the syntax and well-formedness of our program and its specification at each refinement step. The guarantee we get from this correctness level, is a specified and syntactically correct program. The specification does not generate a concrete correctness guarantee (i.e.,

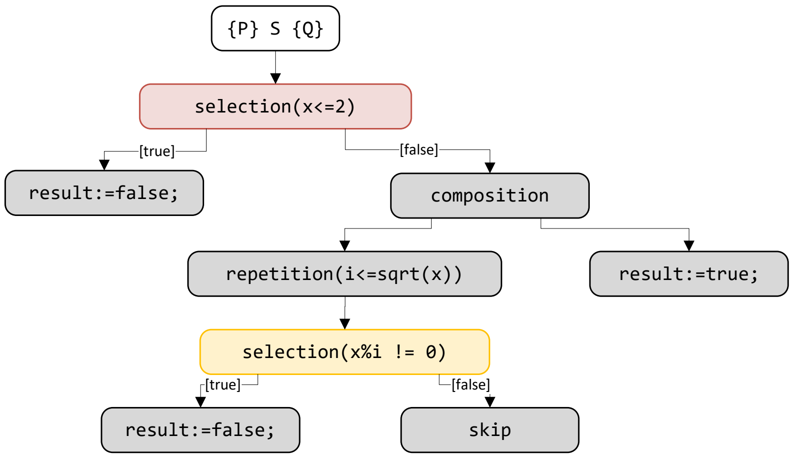


Figure 4.2.: CbC refinement tree of method `isPrime(int x)` (abstracted) (Bordis et al., 2025 (Submitted)). Yellow – Error that is detected in correctness level *tested*. Red – Error that is only detected in correctness level *verified*.

that the program indeed satisfies its specification), as an executable test or proof would, but the intent of the program and its implementation are decoupled, which helps to find first errors in an early implementation phase. Additionally, a formal specification is the basis for many verification approaches, and in our case, it is the basis for the test case generation at the correctness level *tested* and the deductive verification at the correctness level *verified*.

Tool-Support. In theory, writing syntactically correct specifications and programs seems like a simple task. In reality, however, understanding error messages and exceptions can be tedious, especially for a refinement-based approach, such as CbC, that might not be familiar. For CorC, there are some exceptions thrown that are not easily interpretable for new users. For instance, in our example specification of method `isPrime(int x)`, we used a predicate `hasFac(int y)` that is defined outside the CbC program. If there was a syntax error in the definition of this predicate, KeY, the theorem prover in the backend, would throw a `ProofInputException` without providing any further context to the user. Our goal is to improve the output format of these error messages by enriching them with additional knowledge, i.e., additional context information that we gained from working with CorC and KeY. We aim for error messages similar to those in the Rust programming language (Klabnik, 2023), where typical causes are listed to help locate the error.

Level *Tested*. From the previous correctness level, we get a well-formed CbC program in a refinement tree structure, see Figure 4.2. For this example, we show the entire refinement tree, but the test case generation and verification are also applicable to partially defined programs, since each refinement step can be checked individually. In the current development process with CbC, we would now verify that each refinement step satisfies its side condition. If the verification fails, the user can examine the proof tree in KeY, which is not easy to understand, especially for non-experts (Runge et al., 2020). Therefore, we integrate the intermediate correctness level *tested* into the original CbC development process of specification and verification, where test cases are generated from the specified program to provide the user with test results that are easier to understand than open proof goals. We argue that coarse-grained errors, such as using a wrong operator or accidentally swapping the cases of a selection, can be detected more quickly by testing. The correctness guarantee that we obtain from this level of correctness is that the program fulfills its specification for certain concrete input values that have been tested.

Example 4.2. *In Figure 4.2, we show the CbC refinement tree of method* `isPrime(int x)`*, which contains two errors. By generating test cases, we detect the error in the yellow refinement step, where the cases are swapped such that* `false` *is returned if the parameter* `x` *is not divisible by the iteration variable* `i`*. This is a coarse-grained error that leads to a wrong result for many inputs. Therefore, it is easily detectable with test cases that test multiple input values for parameter* `x`*. The user can look at the failed test cases, for example,* `Integer.MAX` *(a prime number) that resulted in* `false` *as output, and then analyze the program and locate the error.*

Tool-Support. Similar to the current verification workflow in CorC, test case generation can be triggered directly in the graphical editor of CorC. After test case execution, the user receives an error report showing the input, output, and expected value of all test cases. In addition, the user can analyze the generated test cases to help locate the error. The input values for the test cases can be generated from the precondition or be obtained from default values for the data types of the parameters. The postcondition is used as a test oracle for the expected result.

Level *Verified*. At the correctness level *verified*, we prove the side conditions of each refinement rule with the theorem prover KeY. This gives us the strongest correctness guarantee for the functionality of our program. However, it is also the most difficult and time-consuming guarantee to achieve. One reason for an open proof goal may be corner cases where the program does not satisfy its specification. As mentioned above, examining the proof tree in KeY requires expertise, which we want to mitigate by generating a counterexample from the proof tree.

Example 4.3. *In Figure 4.2, the error in the red refinement step was not found in the previously generated test cases. Since 2 is the first prime number, the selection guard should be* `x < 2` *instead. The program works in each case except for* `x = 2` *as input (corner case). From the proof tree, we can generate this counterexample that would point the developer directly to this corner case where the program fails the specification.*

Tool-Support. A counterexample can be automatically generated by an SMT solver whenever a proof goal cannot be closed. If the generation of a counterexample was successful, it is displayed in the console of CorC in the form of a value assignment (like in the example above `x = 2`). The developer can then work through the counterexample of value `2` for variable `x` in the specification and program, and determine why in this corner case the program does not satisfy its specification.

Goals of the Three-Level CbC Process. By introducing the third intermediate correctness level *tested*, we aim to create a more flexible and accessible software construction approach that has a lower entry barrier for non-experts in formal verification. We generate more feedback with different complexities for the user at each correctness level. As a result, errors are easier to understand, and tool support is easier to use because less knowledge is required. At the same time, the development process is now more flexible since the user can generate weaker correctness guarantees (*tested* instead of *verified*) for certain parts of the program that are less safety-critical. The option to stay at a lower level of correctness in the development process can also be combined with our currently used roundtrip engineering process, which we proposed in the previous chapter (see Section 3.1.2). This roundtrip engineering process allows the import and export of CbC programs to integrate CbC and the tool CorC into other software development processes. Basically, the second step of the roundtrip engineering process (see Figure 3.2), the construction in CorC, can be separated into our three levels of correctness guarantees, and from each of the individual correctness levels, we can generate Java code – that is either *specified*, *tested*, or *verified* – into the original Java project. Overall, this makes it easier to create larger, (partially) verified software projects in concert with other quality assurance tools and techniques.

4.2. Level 1: *Specified*

The first level of correctness guarantee is the level *specified*. The specification that is provided at this correctness level is the basis for the other two correctness levels, *tested* and *verified*. Specified code has a higher correctness guarantee compared to unspecified code because the intent of the method's behavior is described in a formal but decoupled way (Kelly, 1997). We believe that describing the intended behavior before or during the construction of a program encourages the developer to program in a clearer structure with fewer errors. This "*think before coding*" approach can be compared to test-driven development, where programmers automatically structure their code in a way that it is better testable (Beck, 2015).

For CbC, we define specifications in first-order logic. In CorC, we use the first-order logic fragment of Java Dynamic Logic (Beckert et al., 2016) to be precise. Each method has a contract consisting of a pre- and postcondition pair. This contract is then refined by applying the refinement rules from Definition 2.2 to concrete code. Some of these rules require the developer to further annotate the code by providing intermediate conditions or loop invariant and variant.

```
An [ErrorName] Exception occurred.
This happens when [Reason for error].

[Additional Information if available]

To fix this error, try:
> [Bullet points of potential error fixes]
> ...
```

Listing 4.1: Schema of an error message for a thrown exception in CorC (Bordis et al., 2025 (Submitted)).

The first challenge in this step is to define a well-formed specification and program. For example, in method `isPrime(int x)` from the previous section, we specified the pre- and postcondition using the predicate `hasFac(int y)`. Although the specification is syntactically correct, there may still be issues with the definition of the predicate. Normally, in a textual IDE, the compiler would check this automatically in the background. However, CorC's graphical editor is not designed to compile the code, but was only built to support the CbC development style. Hence, CorC's architecture is not easily adaptable to integrate a well-formedness check for the specifications. Therefore, we still rely on the integrated parser for Java Dynamic Logic in KeY. However, KeY lacks comprehensive error handling. In the following, we describe how we support the developer in CorC in defining well-formed specifications by introducing more informative exception handling for KeY's custom-typed exceptions.

4.2.1. Knowledge-based Exception Handling

As described in Section 2.2.2, in CorC, we generate a proof file containing the side condition of the refinement rule that has to be established. This proof file is then loaded in the background into the deductive verifier KeY. KeY then parses this proof file and throws custom-typed exceptions if there is a parsing error in this file or even somewhere else in the project.

Since the first source of errors is compile-time errors, our goal with knowledge-based exception handling is to help users better understand the error message and provide possible solutions to their problems. We show a schematic form of these error messages in Listing 4.1. Instead of just printing the stack trace (as done by KeY), we provide the name of the exception, the reason for the exception, and a list of possible fixes. This way, the user is directly provided with knowledge that only CorC experts may have. The general form of our error messages is inspired by error messages of Rust (Klabnik, 2023) which gained popularity.

We want to provide additional knowledge about KeY's custom-typed exceptions, since they are not widely known by software developers, and there is not much documentation about exceptions. The additional knowledge that we provide comes from our own experience working with CorC and KeY. In particular, we focus on the following three exception types thrown by KeY:

- ProofInputException
- ConvertException
- PosConvertException

ProofInputException. A `ProofInputException` is thrown when KeY was unable to parse the proof file that has been generated by CorC. In particular, methods and predicates that are called in a specification or in the code usually cause this type of exception. For example, the visibility of a method or typos in its name are common failures. However, also syntactical errors in a specification condition are handled by this exception. A common error here is using the keyword `true/false` in specifications when actually the constant `TRUE/FALSE` should be used. We print the following list of possible fixes for this exception:

- Check if all used predicates are defined in the project's helper.key file
- Check if all used methods are defined in the Java class
- Check the spelling of the method in the CorC diagram and in the Java file
- Check if the method is correctly declared static/nonstatic
- Check the method signature, especially the return type and the parameters and their types
- Check that the syntax of your specification is correct (use TRUE/FALSE instead of true/false)

ConvertException. In the proof file that we generate in CorC, we define the classpath of the project to use methods and predicates that are defined outside the CorC diagram. When KeY loads this proof file, it therefore loads the whole project and tries to parse every Java file that is inside the project. A `ConvertException` is thrown by the parser in KeY if something in the project cannot be parsed correctly. This means that this exception is usually thrown when there is a syntax error in a Java class of the project, even though it is not used in the part of the CorC diagram that should be proven. For our error message, we print the name of the class where the KeY parser fails and suggest checking this file for syntax errors.

PosConvertException. A `PosConvertException` is a subtype of the `Convert-Exception`. It is enriched with a location within a file. In CorC, it is usually caused when a method call in a CorC diagram does not match its defined signature, i.e., parameters are missing or ill-typed. We print the following possible fixes:

- Check the method signature, especially the parameters
- Check the spelling of the method in the CorC diagram
- Consider using a classpath if this is an unresolvable classtype

4.2.2. Implementation

Once the causes and possible solutions for the exceptions are known, the implementation of thsese error messages is quite simple. Whenever KeY loads a CorC project, it checks the whole project for compile-time errors and throws an exception if necessary. We simply catch that exception and overwrite the error message returned by KeY with our knowledge-based error message. As a fallback, we print the original error message from KeY.

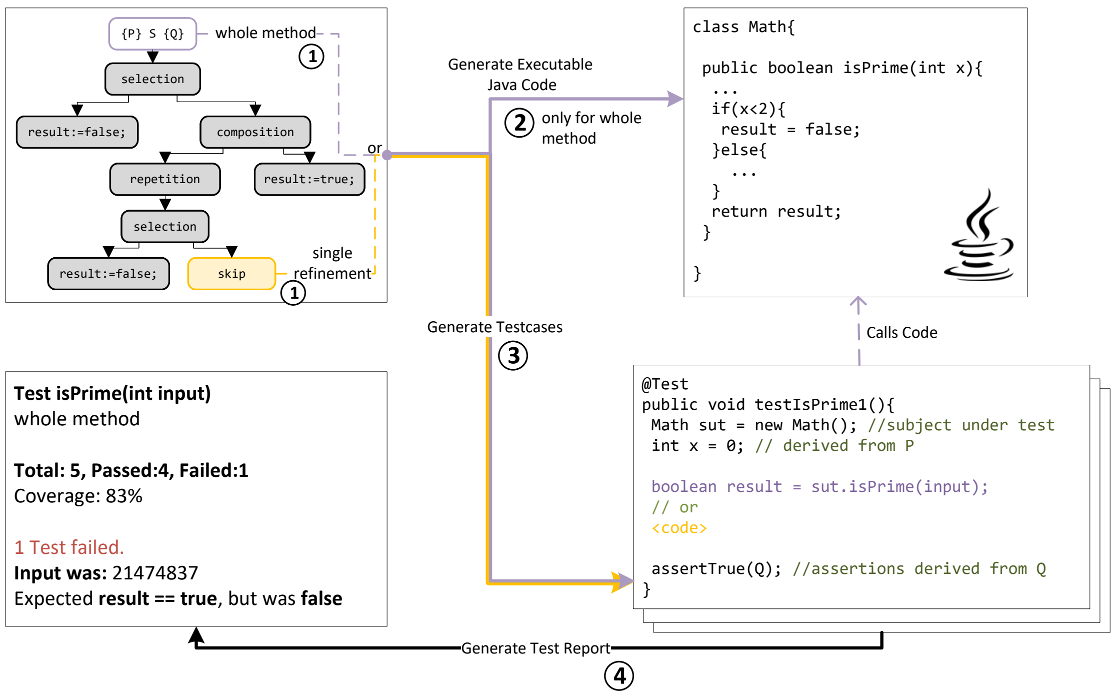
In the future, we plan to adapt CorC's architecture to add our own parser to further improve usability. Our goal is also to provide features, such as code completion and type checking, directly in the CorC editor during development. Since we have had positive experiences with knowledge-based error messages, we want to keep this style and extend the supported exceptions in the future.

4.3. Level 2: *Tested*

In the previous correctness level *specified*, we already developed a specified program that we know is well-formed and free of compile-time errors. In this *tested* level, we want to fix major semantic errors in the code by generating test cases from the already defined specifications. The goal of this correctness level is to use testing as a technique that most software engineers are familiar with, so that the error reports are easy to understand and the reported errors are easy to fix. In this way, only corner cases need to be fixed in the last correctness level *verified*, where understanding the proof requires in-depth knowledge. In the following, we describe how we automatically generate test cases from the pre- and postcondition specifications of the previous correctness level *specified*, the implementation of the test case generation in CorC, and a discussion of related work in the field of test case generation.

4.3.1. Test Case Generation

In Figure 4.3, we give an overview of the steps of the automated test case generation for CbC programs. We start with a (partial) refinement tree containing specifications and program code. Our test case generation can be applied to entire methods or to individual refinement steps to allow testing of partially implemented programs ①. The latter is usually the case when strictly following the CbC approach of iteratively applying a refinement rule and directly checking its side condition. We generate executable Java code from the refinement tree (Bordis et al., 2022a), since we want to use an existing Java testing framework. To test a complete method, we generate the method implementation into its containing class, such that it can be called later by the test case ②. If we want to test a single refinement step (a Hoare triple in the refinement tree of the form `{P} S {Q}`), we generate a partial program code



```
class Math{

 public boolean isPrime(int x){
  ...
  if(x<2){
   result = false;
  }else{
    ...
  }
  return result;
 }

}
```

```
@Test
public void testIsPrime1(){
 Math sut = new Math(); //subject under test
 int x = 0; // derived from P

 boolean result = sut.isPrime(input);
 // or
 <code>

 assertTrue(Q); //assertions derived from Q
}
```

Test isPrime(int input)
whole method

Total: 5, Passed:4, Failed:1
Coverage: 83%

1 Test failed.
Input was: 21474837
Expected **result == true**, but was **false**

Figure 4.3.: Overview of the automatic test case generation (Bordis et al., 2025 (Submitted)).

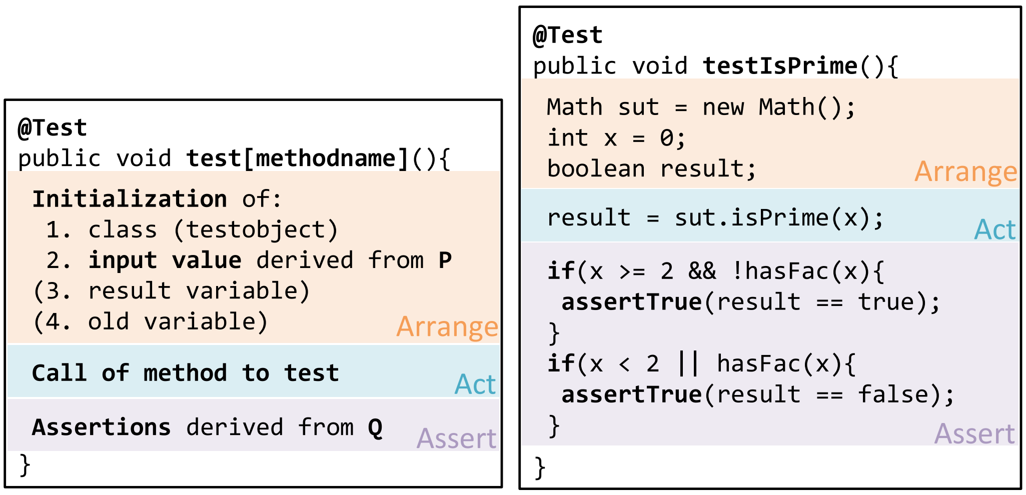
by traversing the refinement tree until we reach statement S. We then generate a test suite consisting of multiple test cases for different input values ③. The exact number of test cases depends on the number of parameters and the strategy that is used to generate inputs. Each test case consists of 1) initializing all required variables (e.g., parameters and test object), 2) executing the test object (calling the generated method or partially generated program code), and 3) checking the assertions that are generated from postcondition Q.

After executing all test cases, we generate a test report ④. We print two different kinds of test reports. First, we print a report for each failed test case with its test input and the failed assertion(s). Second, we print a test end report, which shows the total number of passed, failed, and skipped test cases, as well as the coverage of the postcondition to evaluate the expressiveness of the generated test suite.

Since the generation of JAVA code from CbC programs has already been described in Chapter 3, we will concentrate on the generation of test cases itself and on the test report in this section.

Test Cases

We first describe the general scheme in which we generate test cases and then describe the two main steps of deriving test inputs from preconditions and assertions from postconditions.


Arrange
Act
Assert

```
@Test
public void test[methodname](){
 Initialization of:
  1. class (testobject)
  2. input value derived from P
 (3. result variable)
 (4. old variable)
 Call of method to test
 Assertions derived from Q
}
```

(a) AAA-Schema of a Generated Test Case

```
@Test
public void testIsPrime(){
 Math sut = new Math();
 int x = 0;
 boolean result;
 result = sut.isPrime(x);
 if(x >= 2 && !hasFac(x)){
  assertTrue(result == true);
 }
 if(x < 2 || hasFac(x)){
  assertTrue(result == false);
 }
}
```

(b) Test Case for Method `isPrime(int x)`

Figure 4.4.: Schema of a generated AAA (Arrange, Act, Assert) test case and a concrete example for a generated test case for method `isPrime(int x)` (Bordis et al., 2025 (Submitted)).

Schema of a Generated Test Case. In Figure 4.4, we show the schema in which each test case is generated in Figure 4.4a, and a generated test case for example method `isPrime(int x)` (see Section 4.1 for more details) in Figure 4.4b. As shown in Figure 4.4a, we chose the AAA-schema which is widely used in the field of unit testing (Beck, 2015). AAA stands for *Arrange*, *Act*, and *Assert*, which are the main three steps in a test case.

Arrange. In the first step, *Arrange*, the subject under test and the test inputs are initialized. In our case, we initialize the generated class and derive input values from the precondition `P` of our method to be tested. This step is described in the next subsection. We can usually derive multiple input values from the precondition. We get a test case for each of the input values. In our specifications in CorC, we can use two keywords that we need to handle in this step if they have been used. The keyword `\result` is used to refer to the return value of a method. The keyword `\old(x)` is used to refer to the value of a variable `x` before executing a statement `S` in a Hoare triple `{P} S {Q}`. For both keywords, a variable is initialized for using these values in the next two steps. In the example in Figure 4.4b, we can see that the class `Math` is initialized using a constructor, as well as the input for parameter `x` and the result variable that will be used later to store the return value of method `isPrime(int x)`.

Act. In the next step, *Act*, the method to be tested is called. In the example shown in Figure 4.4b, we can see that method `isPrime(int x)` is called on an object of type `Math`, using variable `x` as actual parameter and storing the returned value in variable `result`. When testing a single refinement step, instead of calling a method, we execute the program directly until the statement in the considered refinement step is reached. This is only possible, if executable code can already be generated from the refinement tree up to the selected refinement step.

Assert. In the last step, *Assert*, we use the postcondition `Q` as a test oracle and derive assertions from it. Since postconditions contain constructs that cannot be used directly in JAVA, e.g., a universal quantifier, we translate the postcondition partly into program logic and assertions, which then check if a certain condition described in the postcondition holds for the inputs. In the example in Figure 4.4b, we split the postcondition

$$\texttt{Q:}(\texttt{x} \geq 2 \land \neg\texttt{hasFac}(\texttt{x}) \rightarrow \texttt{result} = \texttt{true}) \land (\texttt{x} < 2 \lor \texttt{hasFac}(\texttt{x}) \rightarrow \texttt{result} = \texttt{false})$$

in two predicates. The implications are translated into an if-statement with an assertion inside. This step is further described in the next subsections.

From Preconditions to Test Inputs. The selection of input values for test cases is a major challenge in testing, as it determines the coverage of the test object and, thus, the strength of the correctness guarantee (Ammann and Offutt, 2017). There are generally two strategies for generating meaningful test inputs: context-dependent generation, where the test inputs are derived from the specification or the code, or context-independent generation, where default or random values for the given data types independent of the context are used.

We propose a context-dependent strategy with a context-independent alternative. We use two sources of information: 1) the data types of accessible fields and method parameters, and 2) the precondition, which describes the state of the program before a method is executed. To generate test inputs from a precondition, we use an SMT solver. If the precondition is too weak to decide the values of all inputs, we return to defined default values for each input type, with the possibility to define custom test inputs as well.

Context-dependent Test Input Generation. The *Satisfiability Modulo Theories* (SMT) problem extends the SAT problem to include arithmetic, quantifiers, arrays, function calls, and more (Barrett and Tinelli, 2018). The goal is to find a satisfying assignment for a first-order logic precondition with respect to a chosen theory. After translating the precondition into SMT solver syntax, the solver generates an input value for each parameter that satisfies the precondition.

For each variable, the generation is repeated to generate multiple valid values. For each iteration, we add the condition that the variable must not have the value that we already got from the previous rounds. For example, we are looking for an input value for variable `int x` with precondition `P`. In the first round, we give the SMT solver only the precondition `P` and get the input value `1` for variable `x`. In the next round, we call the SMT solver with the condition $\texttt{P} \land \texttt{x} \neq \texttt{1}$ and get another valid value except for `1`. The number of iterations can be chosen by the user. In the end, we have a list of input values for each variable that we use as input for our test cases.

Using an SMT solver to generate test inputs has the limitation that finding satisfying assignments for variables that must have certain characteristics may be difficult or even an undecidable problem. Even if the constraints given by the precondition are linear and the variables are restricted to being integer types, the problem is already NP-hard. For polynomial constraints over integers, the SMT problem is undecidable. Therefore, we define default values if the SMT solver cannot generate a satisfying assignment based on the precondition.

Default Input Values. If the SMT solver-based test input generation was not successful, we have a context-independent strategy where we assign default values for all primitive data types in JAVA (boolean, byte, char, short, int, and long) and string and arrays. Context-independent means that we do not check the precondition. For the primitive data types, we use values that would also be used in boundary value analysis, known from black box testing (Beck, 2015). That means, we use the minimum and maximum values, and values in the middle of the value range. For example, for data type `int`, the default values are `Integer.MIN, -1, 0, 1,` and `Integer.MAX`. We choose these values because values near the boundary of the range are more likely to cause errors (Beck, 2015). Also, `0` is often used as a boundary, and `1` and `-1` are then the two values that are "off-by-one". For strings, we use an empty string, a single-character string, and multi-character strings, including special characters. For arrays, we simply initialize the array with random entries from the defaults of the array's data type. For non-primitive data types, such as instances of self-defined classes, we use JAVAs reflection functionality and load the corresponding class to use a constructor with our default values for the parameters to create an instance of that class for testing. In addition to the context-independent default values, the user can also define custom values for each data type, such that even if the case that the context-sensitive strategy of using the SMT solver fails, context-dependent data can be used as test input.

When testing a method, we usually want to cover all combinations of the input values in the defaults list for all combinations of the parameters. However, in an object-oriented project, there are usually more fields accessible than just the method's parameters. In this case, the number of test cases would increase exponentially with each accessible field. This is also a known problem in variant-rich systems, where combinatorial interaction sampling is used to select the best combinations in the problem space (Yilmaz, 2013). To minimize this, we first determine which fields are used by the method that we want to test. All fields that are not used, but need a value to make the program syntactically correct are then assigned a random value for their data type. For the fields that are used, we generate test cases that cover all combinations of default input values.

From Postconditions to Assertions. In the postcondition, the developer describes the intended behavior of a method with a logical formula. We want to use these postconditions as test oracles to generate assertions that can be automatically checked in a unit test case. Specifications in CORC are first-order logic formulas written in JavaDL syntax (Beckert et al., 2016) with two additional keywords, `\old` and `\result`. With the keyword `\old`, we can refer to the value of a variable before executing a statement. With the keyword `\result`, we can refer to the return value

Operator	CorC Specification	JAVA Code
Return Value	\return	`var result = <method call>;`
Old Value (Value of a variable before execution)	\old(x)	`var old_x = x;`
Comparisons	<, >, <=, >=, ! =, =	`<, >, <=, >=, !=, ==`
Logical And and Or	&, |	`&&, ||`
Logical Implication	$A \rightarrow B$	`if(A){assertTrue(B);}`

Logical Equivalence	$A \leftrightarrow B$	`if(A){assertTrue(B);}`
		`if(B){assertTrue(A);}`

Universal Quantifier	\forall τ x ; A(x)	`for(τ x = τ.MIN_VALUE; x <= τ.MAX_VALUE; x++){`
		`  assertTrue(A(x));`
		`}`

Existential Quantifier	\exists τ x ; A(x)	`boolean exists = false;`
		`for(τ x = τ.MIN_VALUE; x <= τ.MAX_VALUE; x++){`
		`  if(A(x)){ exists = true; break; }`
		`}`
		`assertTrue(exists);`

Table 4.1.: Translation of CORC specifications to assertions for the test cases (Bordis et al., 2025 (Submitted)).

of a method. Since we want to reuse a JAVA test framework for testing, and JAVA does not support all logical operators, we need to translate JavaDL syntax in our CORC specification into JAVA program logic with assertions that can be checked in a unit test. Therefore, we split the postcondition Q into smaller predicates Q_i by splitting Q at its top-level conjunctions. For example, the postcondition of method `isPrime(int x)`

$\mathtt{Q}{:}(\mathtt{x} \geq 2 \land \neg\mathtt{hasFac}(\mathtt{x}) \rightarrow \mathtt{result} = \mathtt{true}) \land (\mathtt{x} < 2 \lor \mathtt{hasFac}(\mathtt{x}) \rightarrow \mathtt{result} = \mathtt{false})$

is separated into the predicates

$\mathtt{Q}_1 : (\mathtt{x} \geq 2 \land \neg\mathtt{hasFac}(\mathtt{x}) \rightarrow \mathtt{result} = \mathtt{true})$

$\mathtt{Q}_2 : (\mathtt{x} < 2 \lor \mathtt{hasFac}(\mathtt{x}) \rightarrow \mathtt{result} = \mathtt{false})$.

The separation into predicates is not only useful for translating the logical specification into JAVA code, but also helps to assess the quality of the generated test suite by later measuring clause coverage in the test report.

In Table 4.1, we list all logical operators in CORC specifications and their translation into JAVA syntax for our generated test cases. For the keywords `\result` and `\old`, we define variables in the *Arrange* part of the generated test case, such that the variables can be used later in the code of the test case. Comparison operators and logical AND and OR can be translated directly into JAVA syntax. However, the logical implication and equivalence, as well as the universal and existential quantifiers, are not part of JAVA's syntax. Therefore, we translate them into program logic. For logical implication of the form $A \rightarrow B$, we use an if-construct with A in the guard and an assertion with B in the then-branch. For logical equivalence, we use a second if-construct with B in the guard and A in the assertion. To translate the universal quantifier $(\forall\ \tau\ x;\ A(x))$ and the existential quantifier $(\exists\ \tau\ x;\ A(x))$, we use a for-loop that iterates from the minimum value of the data type τ of x to the maximum value. For the universal quantifier, we assert the condition $A(x)$ in the body of the loop. For the existential quantifier, we create a variable `exists`, which we initialize with `false` before entering the loop. In the loop body, we set `exists` to `true` when we find a variable that satisfies condition `A(x)` and then exit the loop. At the end, we assert that `exists` must be `true`. For the translations of the existential and universal quantifiers, the data type τ must have an order due to the relational operator that is used in the loop guard. In JAVA, relational operators are supported for the primitive data types except for boolean (chars are compared according to their ASCII code value). However, they are not supported for objects, and consequently our translation does not support objects as data types for τ either.

Test Reports

After executing all generated test cases, we generate two kinds of test reports. First, we generate a test end report that summarizes the results of all test cases and evaluates the test suite against a coverage criterion. Second, we produce a test report for each failed test case, with the goal of providing all necessary information to fix the error.

Test End Report. To summarize the test results, we first collect the statistics on the total number of executed test cases and the number of passed, failed, and skipped test cases. To evaluate the quality of the generated test suite, we additionally measure the clause coverage of the postcondition (Ammann et al., 2003). In dynamic testing, there are various coverage criteria that are used to determine how well the test cases cover the code being tested. For example, statement coverage determines the percentage of total lines of code executed by the test cases (Ammann and Offutt, 2017). For clause coverage, we determine whether each clause (atom without logical connectors) in our postcondition, which we use as a test oracle, has been `true` once and `false` once.

Example 4.4. *In Figure 4.5, we show the test end report for method* `isPrime(int x)` *with the pre- and postcondition*

P:$\texttt{true}$

Q:$(\texttt{x} \geq 2 \land \neg\texttt{hasFac}(\texttt{x}) \rightarrow \texttt{result} = \texttt{true}) \land (\texttt{x} < 2 \lor \texttt{hasFac}(\texttt{x}) \rightarrow \texttt{result} = \texttt{false})$

Since the precondition is too weak to get values for the parameter input, the default values for type `int` *are used (*`Integer.MIN, -1, 0, 1, and Integer.MAX`*). A total of five test cases were generated and executed. Out of these five, only four passed, and one test case failed. In Section 4.1, we introduced the implementation of method* `isPrime(int x)` *and a fault that we can detect with testing, which is located inside the repetition refinement. The repetition checks if an input greater than or equal to* `2` *has a divisor different from* `1` *and itself. The fault was that the cases of the selection inside the repetition were swapped, so that numbers without a divisor returned* `false` *instead of* `true`*. Since all values smaller than* `2` *are processed before*

```
+=======================================+
+ Total tests run: 5, Passed: 4, Failed:1, Skipped:0  +
+=======================================+

>Clause coverage (postcondition): 66% (4/6 clauses)

>Covered clauses (have been true and false) are highlighted:
 x >=2  // true: 2147483647, false: -214748368, -1, 0, 1
 !hasFac(x)  // true: 214748367, false: --
 result == true  // 214748367, false: -214748368, -1, 0, 1
 x < 2  // ...
 hasFac(x) // ...
 result == false  // ...

>Failed test case(s):
 int x = 2147483647 //derived from default list (is a prime)
 ...
 >>>>>assertTrue(result);
```

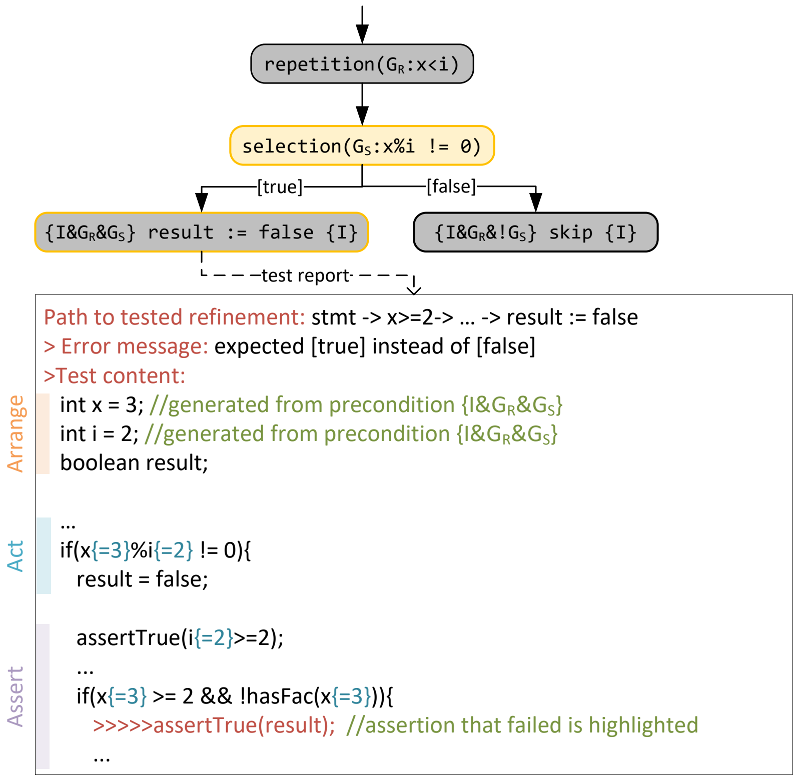
Figure 4.5.: Test end report for method `isPrime(int x)` with all executed test cases and clause coverage (Bordis et al., 2025 (Submitted)).

entering the loop, the first four test cases with the inputs `Integer.MIN, -1, 0,` `and 1` *pass. Only the last test case with input* `Integer.MAX` *fails.*

After these statistics, the clause coverage is computed. The postcondition contains six atomic clauses, which are listed with the clauses highlighted. With the generated test suite using the default input values, we achieved a coverage of 66% (4 out of 6 clauses). Only the clauses `hasFac(x)` *and* `!hasFac(x)` *were not covered because the default list does not contain a value that is not a prime number and that is greater than or equal to two.*

Test Report for a Failed Test. Our goal for test reports for failed test cases is to provide all information necessary to analyze the discovered faults in the program.

Example 4.5. *In Figure 4.6, we show the test report for testing an assignment statement in our example method* `isPrime(int x)`. *At the top of the figure, we show the refinement tree of method* `isPrime(int x)`, *introduced in Section 4.1. The fault that we can find with testing is located in the guard of the selection refinement with swapped cases. After executing the generated test cases, we get the report of the failed test case shown at the bottom of Figure 4.6. First, we print the path in the refinement tree to make it clear, which statement was tested. Second, we print the original error message from the test framework. In this case, an assertion failed because it expected a value of* `true`, *but the actual value was* `false`. *Third, we print the content of the generated test case, which highlights the failed assertion. In the test content, we can see the AAA-schema that we already introduced earlier in this section. In the Arrange part, the parameter* `x` *and the value of variable* `i` *were generated with*


repetition(G_R:x<i)
selection(G_S:x%i != 0)
[true]
[false]
{I&G_R&G_S} result := false {I}
{I&G_R&!G_S} skip {I}
test report
Arrange
Act
Assert

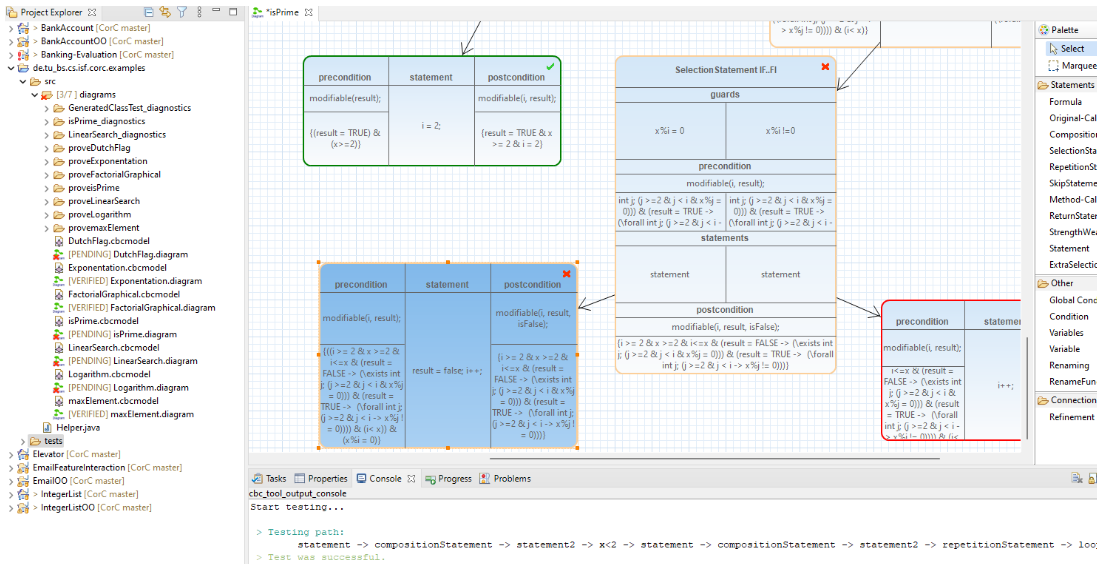
```
Path to tested refinement: stmt -> x>=2-> ... -> result := false
> Error message: expected [true] instead of [false]
>Test content:
 int x = 3; //generated from precondition {I&G_R&G_S}
 int i = 2; //generated from precondition {I&G_R&G_S}
 boolean result;

 ...
 if(x{=3}%i{=2} != 0){
   result = false;

   assertTrue(i{=2}>=2);
   ...
   if(x{=3} >= 2 && !hasFac(x{=3})){
     >>>>>assertTrue(result);  //assertion that failed is highlighted
     ...
```

Figure 4.6.: Test report for a failed test case of method `isPrime(int x)` with `x = 3` (Bordis et al., 2025 (Submitted)). The problem is swapped cases in the guard of the selection statement.

the SMT solver from the invariant `I` *of the repetition refinement and the guards* G_R *and* G_S *from the selection and repetition refinements. In the Act part, the partial code is printed instead of a method call, since we are testing a single refinement in this case. Finally, in the Assert part, the postcondition* `I` *of the assignment is translated into program logic with assertions that must be* `true` *to pass the test case. Here we can see that the assertion* `assertTrue(result);` *failed (marked red) because the result was set to* `false` *in the Act part of the test case because the guard of the selection was wrong. In the Act and Assert parts, we also put the value of the variables* `x` *and* `i` *in blue curly braces (e.g.,* `x{=3}`*) after each occurrence of the variable to make it easier to locate the fault during debugging.*

4.3.2. Implementation

We implemented test case generation in our tool CorC, which we introduced in Section 2.2.2. We modified the underlying CbC meta-model by adding the correctness state *tested*. In the graphical editor, which is used to implement programs in the form of refinement trees, the borders of the refinement steps are then colored either red for neither tested nor verified, yellow for tested, or green for verified. Test case generation can be triggered in the context menu either by clicking on a single refinement step or anywhere in the editor for a complete method. Test cases are automatically generated



```
cbc_tool_output_console
Start testing...

 > Testing path:
       statement -> compositionStatement -> statement2 -> x<2 -> statement -> compositionStatement -> statement2 -> repetitionStatement -> loo
 > Test was successful.
```

Figure 4.7.: Screenshot of CorC with test case generation. The yellow border in the refinement tree denotes that the refinement step has been *tested*. In the console, the test report is printed.

and executed in the background. The results are then printed in the form of a test end report and test reports for failed test cases in the CorC console. A screenshot of the graphical editor with a test report in the console is shown in Figure 4.7.

The three main steps in our test case generation are 1) code generation, 2) test case generation (determining test inputs and assertions from the postcondition), and 3) test report generation (see Figure 4.3). The generation of executable Java code has already been described in Chapter 3. We use TestNG (Béust, n.d.) as a test framework because it is well-known to Java developers and offers more options to parameterize its use compared to JUnit. To derive test inputs for the generated test cases, we use the SMT solver Z3 for the context-dependent test input generation, because Z3 is also used by KeY and can also be used for counterexample generation in the next correctness level *verified*. In case that Z3 cannot determine input values from the precondition, we have defined a list of default values for each primitive data type in Java and strings. The lists can also be modified by the user. To translate the postcondition into program logic and assertions, we implemented a parser for CorC's specification language, which is written in JavaDL syntax with the additional keywords `\result` and `\old`. The resulting AST can be converted to Java code as described in Table 4.1.

For the test end report and the test report for single failed test cases, we collect the necessary information and print it in the CorC console. For the test end report, we print the number of failed, passed, and skipped test cases from TestNG. To measure the clause coverage, we instrument the code with boolean flags, which are set to true if a clause in the test case was true when the assertion of the clause was checked in the test case. For the test report of a failed test case, we print the error message from TestNG and print the generated code and the test case itself. We modify the original test case code by cleaning it from the instrumentation of the clause coverage, inserting the parameter values in curly braces, and highlighting the failed assertion in red.

4.3.3. Discussion

Automated test case generation is a well-researched field, with most research focussing on the generation of test inputs rather than test oracles. Test oracles can be derived from a behavioral specification, such as structured documentation (Peters and Parnas, 1998), a formal model (Utting et al., 2012), or formal specifications written in a logic-based specification languages, such as JavaDL, as we have done. The underlying idea of each approach is the same: a test oracle is generated for each unit based on that unit's specification, tests are executed with input data, and a test is passed if the execution of the program with the input data does not violate the unit's specification.

There are different categories of test input generation: *random-based* (test inputs are generated randomly), *optimization-based* (test inputs are generated iteratively based on code coverage criteria), *symbolic execution-based* (symbolic execution is used to determine test inputs that reach a particular part of the program), *specification- or model-based* (test inputs are generated using constraint solving on the specification or model), and *verification-based* (test inputs are generated from failed verification attempts) (Edvardsson, 1999). Our approach can be categorized as specification-based test input generation. In the following, we describe the most related of the existing automated test generation tools in this category.

There are several tools that support specification-based test case generation, such as JMLUnitNG (Zimmerman and Nagmoti, 2011), TestEra (Khurshid and Marinov, 2004), and AutoTest (Leitner et al., 2007). The test case generation tool JMLUnitNG is an extension of the tool JMLUnit (Cheon and Leavens, 2002) that generates test cases for Java programs specified with JML (Leavens and Cheon, 2006). It generates test input based on default values or user-defined values for input parameters, and uses the postcondition as a test oracle to generate assertions with the test framework TestNG. The test case generation tools TestEra and AutoTest both support strategies for automatically generating context-sensitive test input. TestEra uses a SAT solver to generate context-sensitive input data and is applicable to Java programs with Alloy (Jackson, 2016) specifications. As TestEra uses a SAT solver, it does not support arrays. However, TestEra supports testing complex data structures, such as trees. AutoTest targets the automatic generation of test cases for Eiffel (Meyer, 1988) programs and implements different strategies to derive context-sensitive test input.

Other tools, such as BZ-TT (Ambert et al., 2002), JML-Testing-Tools (Bouquet et al., 2005), and UniTesK (Bourdonov et al., 2002) analyze the specification or model to compute equivalence classes (also called partitions) of the inputs, where each input in the same equivalence class is assumed to behave the same. Once the equivalence classes are computed, constraint solving or model finding is used to find concrete test data in each equivalence class.

All of the above-mentioned tools use similar techniques to derive test inputs and use the postcondition as a test oracle. Our goal is to implement an automatic test case generation approach tailored to the fine-grained refinement process of CbC in our tool CorC. The main difference to the approaches mentioned above is that we support test case generation for individual refinement steps/statements and not only for complete methods or classes. To implement this, we adapted techniques that have already been implemented in the tools above, such as default input values and an input selection from JMLUnitNG, or an SMT-based context-sensitive strategy similar to TestEra. We focus on keeping the approach lightweight and user-friendly by including detailed error reports.

4.4. Level 3: *Verified*

In the previous correctness level *tested*, we generated test cases that give us the guarantee that our program is correct wrt. the specification for certain tested input values. With this correctness level *verified*, we can give a stronger correctness guarantee in the sense of total correctness. Given a program S, a precondition P, and a postcondition Q, in any state that satisfies precondition P, we guarantee that if the program S is executed, it terminates in a state described by postcondition Q. We guarantee total correctness by developing our programs using CbC as described in Section 2.2. In our tool CorC, we use the deductive program verifier KeY to prove that the side conditions of the CbC refinement rules hold.

In previous work (Runge et al., 2019), we found that having finer-grained specifications in the CbC approach leads to a reduced proof complexity for KeY compared to post-hoc approaches where a complete method is specified and verified as a whole. Therefore, using CbC, the proofs are already more likely to be automatically closable compared to post-hoc approaches. However, in that work we have only examined single algorithms. For more complex programs, verification may not be automatable and there may still be errors in the program due to corner cases that are not obvious to the developer if a proof cannot be closed. Locating the error by examining the proof tree in KeY is difficult, especially for developers without knowledge of formal methods and KeY. Therefore, we want to support developers in solving these cases when a proof is not found in CorC by generating counterexamples.

4.4.1. Counterexample Generation

A counterexample is a variable assignment that satisfies the precondition `P` of a Hoare triple `{P} S {Q}`, but not the postcondition `Q` after the program S has been executed. In other words, counterexamples provide an exemplary input that does not satisfy the postcondition when the program is executed. SMT solvers can be used to generate counterexamples in the case that an SMT problem cannot be solved. The deductive verifier KeY, which we use in CorC to verify the correct application of CbC refinement rules, already defines an interface for using an SMT solver on top of its deductive verification capabilities. We use this interface to translate an open proof goal in a proof of an side condition done with KeY into an SMT problem and use the SMT solver Z3 (De Moura and Bjørner, 2008) to generate such a counterexample. Since the output of Z3 is not meant to be read by humans, we remove information that is not related to the input of the program and translate the functions in the syntax of SMT-LIB (Barrett et al., 2010) into more commonly known assignments to give a meaningful error message.

Example 4.6. *In Section 4.1, we introduced method* `isPrime(int x)` *as an example. In Figure 4.2, we show its CbC refinement tree with an error in the refinement with the red border. If we check this refinement step with CorC, the proof ends with the open goal shown in Listing 4.2. This means that the proof could not be closed for the case* `x = 2` *and the expected result* `true`*. In our example, this open goal is easy to understand, but this is not always the case. In Listing 4.3, we show the counterexample in SMT-LIB (Barrett et al., 2010) that is generated from the open goal in Listing 4.2. Z3 uses a symbolic domain to model the behavior of a program. It defines the universe in Lines 3 and 4 by defining several symbolic values and their definitions as functions in Lines 5–12. The constraint in Lines 13–20 ensures that each object in the universe is assigned exactly one of the symbolic values. The function definitions in Lines 22–32 assign the values of the counterexample to the program variables. Since the most important information of the counterexample is hidden in Lines 22–32, we perform a cleanup on the generated counterexample to produce an output that is easier to understand for the developer. In Listing 4.4, we show the cleaned output, which we print to the console of CorC. We remove the model of the universe and filter out variables that are most likely irrelevant to understanding the counterexample. We also translate the s-expression syntax of SMT-LIB into more commonly known assignments.*

4.4.2. Implementation

We integrate Z3 as an SMT solver in CorC and use KeY's SMT interface to generate the counterexample from an open proof goal. From the verification with KeY, we get a list of open goals if a proof cannot be closed. We go through the list of open proof goals and translate each goal separately into an SMT problem and pass it to Z3. Z3 returns either `SAT` (if it could close the goal) or `UNSAT` (otherwise). In the case that Z3 returned `UNSAT`, it tries to generate a counterexample. We iterate through the list of open proof goals until a counterexample is found or all open goals are processed. If a counterexample could be generated, we clean the original output of Z3 and print it to the console of CorC.

```
x = 2, result = TRUE, wellFormed(heap) ==>
```

Listing 4.2: Open goal after verifying method `isPrime(int x)` in KeY (Bordis et al., 2025 (Submitted)).

```
sat
(
;; universe for u:
;;   u!val!0 u!val!2 u!val!5 u!val!4 u!val!1 u!val!3
;; -----------
;; definitions for universe elements:
(declare-fun u!val!0 () u)
(declare-fun u!val!2 () u)
(declare-fun u!val!5 () u)
(declare-fun u!val!4 () u)
(declare-fun u!val!1 () u)
(declare-fun u!val!3 () u)
;; cardinality constraint:
(forall ((x u))
(or (= x u!val!0)
(= x u!val!2)
(= x u!val!5)
(= x u!val!4)
(= x u!val!1)
(= x u!val!3)))
;; -----------
(define-fun TRUE_6 () u u!val!1)
(define-fun heap_9 () u u!val!0)
(define-fun dummy_Heap_13 () u u!val!3)
(define-fun result_5 () u u!val!1)
(define-fun x_2 () Int 2)
(define-fun dummy_boolean_12 () u u!val!2)
(define-fun dummy_int_11 () Int 0)
(define-fun type_of_Heap_7_8 ((x!0 u)) Bool true)
(define-fun type_of_boolean_3_4 ((x!0 u)) Bool true)
(define-fun wellFormed_10 ((x!0 u)) Bool true)
(define-fun type_of_int_0_1 ((x!0 u)) Bool false)
)
```

Listing 4.3: Counterexample generated by Z3 (Bordis et al., 2025 (Submitted)).

```
[Int x_2 =  2]
[Int dummy_int_11 =  0]
```

Listing 4.4: Cleaned counterexample printed in CorC (Bordis et al., 2025 (Submitted)).

4.5. Evaluation

In this chapter, we present a three-level process for Correctness-by-Construction program development, with the overall goal of making the construction of provably correct software more accessible to non-experts in the field of formal methods. Our development process includes three increasing levels of correctness guarantees. In our tool CorC,[1] the developer is supported by different techniques that help them achieve the corresponding level of correctness. In this section, we perform a quantitative evaluation of our three-level process and a qualitative evaluation based on a usability refinement study to gain feedback on the tool support at each correctness level and to provide suggestions for improvement.

4.5.1. Research Questions and Subject Systems

We are investigating two research objectives. First, we want to evaluate our three-level process as a whole and assess the effectiveness of introducing an intermediate correctness level of testing in between the standard specification and verification correctness levels. To determine the effectiveness of the process, we introduce defects by mutation into our subject systems and measure at which correctness level the defect is detected. Second, we want to evaluate the three correctness levels in isolation and find out whether the tool support at each correctness level is usable and what can be improved. Therefore, we conduct a qualitative usability refinement study that provides detailed feedback on the tool support at each correctness level. In particular, we address the following research questions:

RQ1 (effectiveness): How well does our three-level process work to locate defects?

RQ2 (usability refinement): How can the tool support in CorC be further improved to support developers in achieving each level of correctness?

We use eight subject systems for the evaluation, which have been used as subject systems in previous evaluations of CorC (Bordis et al., 2022a; Runge et al., 2019), and other automated verifiers (Thüm et al., 2019; Knüppel et al., 2020a; Thüm et al., 2014b; Scholz et al., 2011). Six of the subject systems are single algorithms from Kourie and Watson (2012) and the other two are software product lines (Apel et al., 2013a). We chose these subject systems because they are typical applications for CorC (Runge et al., 2019; Bordis et al., 2020b; Bordis et al., 2023b) and use CorC's most important features. The algorithms are small programs that are easy to understand but have a certain complexity in the implementation and specification, which makes them ideal for the usability refinement study and the mutation-based analysis. The software product line subject systems additionally use object-orientation and variability constructs, which add another layer of complexity. We show some chracteristics that describe the subject systems in Table 4.2.

[1] https://github.com/KIT-TVA/CorC

	#Diagram nodes in CbC-tree	#Classes	#Methods	#Features
Linear Search*	5	-	1	-
Max. Element*	9	-	1	-
Exponentiation	7	-	1	-
Log. Approx.*	5	-	1	-
Dutch Flag*	8	-	1	-
Factorial*	5	-	1	-
Integer List	47	1	5	5
Bank Account	88	3	13	6

Table 4.2.: Characteristics of the subject systems (Bordis et al., 2025 (Submitted)). All subject systems are used for the mutation-based analysis, and subject systems with an asterisk (*) were used for the usability refinement study. The number of diagram nodes in CbC-tree is a metric for the size of a CbC program, similar to lines of code for textual programs. The number of features is a software product line-specific metric, provided only for the product line subject systems.

4.5.2. RQ1 - How well does our three-level process work to locate defects?

We answer *RQ1* quantitatively with a mutation-based analysis that mimics small defects typically introduced during development. First, we mutate the code of our subject systems to introduce a defect. Second, we independently run our test case generation from the correctness level *tested* and the verification from the correctness level *verified* to determine the percentage of defects that can already be found at the correctness level *tested*. This percentage gives us insights into the usefulness of executing generated test cases. The higher this value is, the more defects can be found with test case generation, which has the advantage of being more understandable with concrete inputs for the test cases that failed.

Mutation-based Analysis

For the mutation-based analysis, we have integrated μ*Java* (Ma et al., 2006) into CorC. In the properties view of the tool, individual mutation operators can be selected and applied to the code of a Hoare triple. For example, there is a mutation operator that randomly changes the multiplication operator in the code `a * b` to another arithmetic operator. In general, each mutation operator introduces a small change in the code that simulates careless errors introduced by the developer (DeMillo et al., 1978; Jia and Harman, 2011; Ammann and Offutt, 2017). We give a complete list of supported mutation operators in Appendix B.0.1.

For the evaluation, we mutate each assignment refinement and each guard of a repetition and selection refinement in our subject systems to obtain one mutant each. These resemble the parts in a CbC program that would be mutated in a textual Java program, where the mutation operator is usually applied to a complete method

	Found Defects in %		#Mutations
	Level *tested*	Level *verified*	
Linear Search	66.67	100	3
Max. Element	60	100	5
Exponentiation	80	100	5
Log. Approx.	75	100	4
Dutch Flag	60	80	5
Factorial	25	100	4
Integer List	88	100	25
Bank Account	73	100	37
Total	73.86	98.86	88

Table 4.3.: Results of the mutation-based analysis show that 73.86% of the defects were found in the correctness level *tested* (Bordis et al., 2025 (Submitted)). The mutants are generated by mutating each assignment statement, repetition guard, and selection guard once with an applicable mutation operator.

where a random statement is changed. Unlike other studies that have performed a mutation-based analysis on textual JAVA programs (Knüppel et al., 2021), we mutate each statement in isolation, as this better fits the refinement-based structure of CbC programs. The mutation operator is chosen randomly from the applicable mutation operators. For example, to mutate the assignment `a += b - c;` in the code of a Hoare triple, either the difference operator subtracting `c` from `b` or the assignment operator += is mutated. Other mutation operators are not applicable to this particular statement. Consequently, we mutate our code in a more targeted way, but the set of applicable mutation operators is smaller compared to mutation-based analyses performed on complete methods. The number of mutations for each subject system is shown in Table 4.3.

For the evaluation, we generate and execute the test cases (correctness level *tested*) and verify the refinement (correctness level *verified*) after each mutation. For both correctness levels, we get a result (*success* or *failure* of the test/verification). Thus, *success* means that the correctness level did not detect the defect introduced by the mutation, and *failure* means that it was detected, i.e., in our case *failure* is the positive result, since we want to detect the defects. With this, we can calculate the percentage of defects found in both correctness levels. We assume that the correctness level *verified* detects every failure because all corner cases are covered. The only exceptions are those where the specification was too weak. With the percentage of defects detected at the correctness level *tested*, we can estimate the usefulness of this correctness level in the proposed process in our tool, assuming that it is easier for a software developer to find a defect using a test case than using the proof tree.

Results and Discussion

In Table 4.3, we show the results of the mutation-based analysis per subject system and in total. We give the percentage of defects found that were introduced by the mutation at the correctness level *tested* and correctness level *verified*, and the number of mutations that have been performed. Besides *passed* and *failed* as results for the test cases at the correctness level *tested*, in five cases the mutation changed the program, so that it no longer terminates, causing the test to timeout. We interpret a failed result as a test case that found the defect and a passed or timeout result as a test that did not find the defect. We only consider *closed* and *open* proofs, where *closed* means that the verification did not find the defect and *open* means that it could find the defect.

In the second column of Table 4.3, we can see that the verification found almost all of the defects introduced by the mutation. This shows that the defects have changed the behavior of the code in such a way that it no longer conforms to the specification. The exception is one mutation in the *Dutch Flag* subject system. In this case, the specification was too weak, and therefore the proof was still closable (cp. (Knüppel et al., 2021)). After strengthening the specification, we were unable to close the proof with the mutated statement. The percentage of defects found at the correctness level *tested* ranges from 25% to 88% per subject system. However, the percentage of 25% for the subject system *Factorial* was the result of one out of four mutations, which mitigates the impact of that subject system. In total, 73.86% of the defects could be located at the correctness level *tested*.

Looking at the mutants that were not detected at the correctness level *tested*, we observed that they were mainly changes in the control flow of the program, i.e., the conditions in the guards of repetitions or selections. Conversely, mutated assignments involving variables used in the postcondition were detected very well by our generated test cases. This is not surprising, since the specifications describe the program state and values of variables in a detailed way. For changes in the control flow, corner cases can be created where only a single input variable causes an incorrect behavior. For example, if a guard is mutated from `x > 0` to `x >= 0`, then an observable change in the behavior is only visible for `x == 0`. For any other value of `x`, the behavior remains the same. The generation of input values for the generated test cases depends either on the precondition (if the SMT solver can find a state that satisfies the precondition) or on a set of default values based on the data types of the parameters. In both cases, the input value that causes the input variable `x` from the example to be 0 may be missing. However, since many of the mutations in the guard were caught by our generated test cases, we are confident that we are generating plausible input parameters. For the remaining cases, we rely on the correctness level *verified* to catch the errors.

In summary, we answer RQ1 *How well does our three-level process work to locate defects?* positively, with an average of 73.86% of located defects at the correctness level *tested* and 98.86% of located defects at the correctness level *verified*. As expected, all defects, except for the one case where the specification was too weak, could be located at the correctness level *verified*. This underlines the need for verification to create functionally correct programs. However, we still argue that locating the defect at correctness level *verified* is more challenging, especially when looking at the proof tree. At the correctness level *tested*, the generated test cases were able to locate the defect in 73.86% of the cases, which means that in 73.86% of the cases where a small defect is introduced, the user no longer needs to analyze the proof tree, but can generate test cases with executable code and concrete input values.

4.5.3. RQ2 - How can the tool support in CorC be further improved to support developers in achieving each level of correctness?

To answer *RQ2*, we perform a qualitative usability refinement study with few, but well-selected participants. The usability refinement study consists of a practical experiment and a structured interview for each correctness level. In the practical experiment, the participants solve one task with the techniques that are implemented in CorC to help the developer reach the correctness level and one with the original version of CorC. In particular, we investigate the usefulness of knowledge-based error messages at the correctness level *specified*, the test case generation at the correctness level *tested*, and the counterexample generation at the correctness level *verified*. The focus is on retrieving suggestions for improvement of usability in the tool support since we are working on a research prototype.

We designed our usability refinement study to generate holistic insights through observation of the participants' behavior in the experiment and a qualitative personal opinion in the interview. Our goal is to collect in-depth details about the user interaction with the tool support in each correctness level, about the motivation and obstacles of the participants during the practical experiment, about how the implemented techniques are integrated into the workflow of a CorC user, and about their suggestions for improvement. We therefore decided to keep the number of participants small, such that we can concentrate on each participant individually and interact with them personally. This gives us higher quality results compared to a quantitative user study with many participants. Additionally, we selected participants from our target group, i.e., software developers that are non-experts in the field of software verification, to generate evidence for our hypothesis that our concept and implementation make software verification applicable for non-experts in the field. However, the main focus lies on getting feedback and suggestions for tool improvement.

Usability Refinement Study

For the usability refinement study, we perform an independent experiment for each correctness level of correctness guarantee to assess the usefulness of each correctness level as implemented in CorC in isolation. Each experiment consists of two verification tasks where one task should be solved with the original version of CorC and the other one with the modified version of CorC that offers tool support for the corresponding correctness level. After each experiment, we performed a structured interview with open-ended questions in which participants reported on their experience. The entire usability refinement study is conducted with each participant in a separate meeting.

Participants. For our qualitative usability refinement study, we considered people with experience in software development, but only little to medium experience with software verification. We additionally required the participants to have already worked with CorC. That is, current and former computer science students that used CorC during courses or developed parts of CorC in the past. We decided to restrict the selection of participants to people with experience in CorC because formal verification with CorC is not regularly taught in computer science studies, and our experiments required a certain level of proficiency with the tool. We included a block of self-assessment questions in the interview part of the usability refinement study to verify that the participants meet our target group as described above. The participants voluntarily attended the usability refinement study and knew that the usability refinement study did not affect their studies if they were still enrolled as students. In total, we had five participants that were randomly divided into two groups: For each correctness level, Group 1 solved Task A with the original version of CorC and Task B with the modified version. Group 2 solved the same tasks, but with the other version instead (i.e., Task A using the modified version and Task B with the original version).

Setup. We installed two versions of CorC on one machine: 1) the original version of CorC and 2) the version that implements all the techniques presented in this chapter. We conducted the usability refinement study with each participant in a separate meeting. The participant used TeamViewer[2] to connect to our machine that runs both versions of CorC remotely. The participant then interacted with CorC as if it was installed on their own machine, and we observed their actions. Furthermore, we used BigBlueButton[3] to verbally communicate with the participant. This allowed us to communicate with the participant during the experiment and provide help and explanations where needed. At the beginning of the usability refinement study, participants were given a short introduction to CorC, to the new features in each correctness level, and to the tasks of the study.

Tasks. For the experiments, we created five tasks in which the participants had to find a defect in one of the subject systems. There is one task for the correctness level *specified* and two tasks each for the correctness levels *tested* and *verified*. In

[2] https://www.teamviewer.com/de/
[3] https://bigbluebutton.org/

Table 4.4, we give an overview of all tasks, including the description of the defects that we introduced. For the experiment at the correctness level *specified*, the participants had to find a syntax defect with the improved error messages. At the correctness level *tested*, the participants had to locate a defect in Tasks 2.1 and 2.2, one only with test case generation and the other one only using verification. Group A solved Task 2.1 with test generation and Task 2.2 with verification. Group B did it the other way around to balance different difficulties of the tasks. For the experiment at the correctness level *verified*, the participants had to locate a defect in Tasks 3.1 and 3.2, one only with counterexample generation and the other one without. Group A solved Task 3.1 with counterexample generation and Task 3.2 without. Group B did it the other way around.

Interview. We performed a structured interview with mainly open questions. To assess the opinions about the usability and suggestions for improvements for each level of correctness in CorC, we asked the same questions about the technique of that correctness level after each experiment. We also included general questions to assess the proficiency of the participants with the tools, the subject systems, and the techniques, and how realistic the defects in our tasks were. The list of questions that was asked is shown in Figure 4.8.

With questions **Q1 − 4**, we want the participants to give a self-assessment about their proficiency with the tools, techniques, and algorithms in the tasks before solving them. This helps us to check whether our participants are part of our target group, i.e., software developers that are non-experts in software verification. Furthermore, we can use the participants' answers to better rate their performance during the experiments. We ask questions **Q5 − 8** after each experiment to get the participant's first opinion on using the new technique in the tool CorC. As the techniques are new features of CorC, we especially focus on gathering suggestions for improvements that can be added to CorC in the future. After the last experiment, we ask questions **Q9 & 10** to provide evidence that the tasks that we artificially created resemble realistic defects that have already been experienced by the participants in their time working with CorC. In Appendix B.0.2, we show the protocols of the interviews.

Results

Proficiency of the participants (**Q1 − 4**). The experiences with CorC and KeY ranged between three months and some years of working with these tools. In general, the participants felt comfortable using both tools, but some felt more proficient using CorC than KeY. Out of all algorithms that we used in the tasks, *Linear Search* and *Max. Element* were known the best, while the others were merely known, or at least the implementation was unknown. All participants had experiences with unit testing, but none of them had ever used counterexample generation or test case generation before.

Tasks	Level	Subject System	Description of the Defect
Task 1	*specified*	Factorial	The implementation of *Factorial* uses a helper function that is defined externally in a JAVA class. The defect is a syntax error in that external definition. This is a common mistake, as during verification the whole project is loaded and checked for syntactical correctness.
Task 2.1	*tested*	Linear Search	In the *Linear Search* algorithm, there is a loop that iterates through an array as long as the element that is searched for has not been found. We altered the comparison operator in the guard of the loop from unequal (`A[i]!=x`) to equal (`A[i]==x`).
Task 2.2	*tested*	Maximum Element	The *Maximum Element* algorithm searches for the biggest value in an integer array. Therefore, it uses two indices, `i` and `j`, where index `i` iterates through the array and index `j` is the current maximal element. The defect we introduced for this task is that we returned index `i` instead of index `j` at the end of the algorithm.
Task 3.1	*verified*	Dutch National Flag	The *Dutch National Flag* algorithm is a special sorting algorithm that sorts an array with entries of the three colors red, white, and blue into the correct order of the Dutch national flag. The algorithm uses a selection to differentiate between different cases. We swapped two of the bodies in that selection.
Task 3.2	*verified*	Logarithm Approx.	In the algorithm that approximates the *Logarithm* function, we deleted an assignment that increases the loop index in the loop body.

Table 4.4.: Overview of the tasks that the participants solved during the practical experiment of the evaluation (Bordis et al., 2025 (Submitted)). In each task, a defect had to be located with or without the implemented tool support in the corresponding correctness level.

General questions before the first experiment

Q1: To what extent have you used CorC and KeY before?

Q2: Are you familiar with the algorithms in the tasks?

Q3: Do you have experience with software testing?

Q4: Do you have experience with counterexample generation?

After each experiment

Q5: Were/Was the [error messages/test case generation/counterexample generation] useful to locate the defect?

Q6: Did the [error messages/test case generation/counterexample generation] speed up debugging?

Q7: How can we improve [error messages/test case generation/counterexample generation]? Are there any features missing?

Q8: Would you use [error messages/test case generation/counterexample generation] instead of the original version of CorC?

General questions after the last experiment

Q9: Did the defects in the tasks seem realistic to you?

Q10: Did you see any of these defects before when using CorC?

Figure 4.8.: List of questions that was asked in the interview of the usability refinement study.

Plausibility of the tasks (**Q9 & Q10**). All participants confirmed that the defects that we introduced for the tasks were realistic and had been experienced before when using CorC. One participant claimed that the defect in the *Logarithmic Approximation* algorithm was too obvious, but still realistic.

Error messages at correctness level specified (**Q5** − **8**). We got positive feedback from all participants for the improved error messages. Two participants said that the error messages transport the knowledge that only experienced developers in CorC have. One participant suggested extending the output, such that it contains more details, such as the complete stack trace.

Test case generation at correctness level tested (**Q5** − **8**). All participants gave positive feedback for test case generation. They mentioned that it helped them finding the error more quickly compared to using only KeY at correctness level *verified*. One participant explicitly mentioned that the concrete input values helped them to run through the program in their head to locate the defect in the considered task, supporting our hypothesis that it is easier to interpret test cases than a proof tree in KeY. This is also confirmed in the results of the experiment: With KeY, two out of five participants needed help to locate the defect in their task. Using test case generation, only one participant needed help to understand the path that is printed out in the console, which is not directly related to locating the defect. The participants mentioned some ideas on improvements in the output, such as the graphical presentation of the paths in the CorC-diagram for partially generated test cases, marking which Hoare triples have been tested, showing which assertion in the test case belongs to which postcondition, and a summary of the failed and passed test cases. In general, we conclude that the output format and visualization in the graphical editor of CorC need to be improved.

Counterexample generation at correctness level verified ($\mathbf{Q5-8}$). The counterexample generation was rated with mixed opinions: Three out of five participants rated it as useful. While one participant emphasized that it helped them a lot to locate the defect, other participants criticized that it takes time to get familiar with the syntax of the output, and even then it is too hard to understand. They would rate it as helpful once the output is more readable.

Discussion

We rate the proficiency of our participants to be on the level of junior software engineers, with several years of experience in programming and testing throughout their studies. Additionally, they are comfortable using CorC, which makes them capable candidates for this usability refinement study.

Given the answers in the interviews, we were able to deduce many ideas for improvements in the implementation of CorC for the different correctness levels. We already implemented some ideas, such as introducing a third state to mark Hoare triples as tested, as described in Section 4.3.2. Other ideas that are more time-consuming to implement will be targeted in the future. We especially want to focus on improving the output of the generated counterexamples, as this has been criticized the most. In recent work (Kodetzki et al., 2025a), we experimented with an LLM-based explanation of the counterexample output, which we might consider as a solution to address this problem.

4.5.4. Threats to Validity

In this section, we discuss threats to validity for the quantitative and qualitative evaluation. In the quantitative evaluation, we used a mutation-based analysis to determine the effectiveness of our three-level process. In the qualitative evaluation, we performed a usability refinement study with experiments and interviews to assess the usability of the concepts implemented in CorC and generate ideas for improvements.

RQ1 – Quantitative Evaluation

Internal Validity. Our chosen methodology to generate mutants lacks compatibility with other mutation-based evaluations. To generate mutants for the evaluation, we used a prominent tool, but did not use it on complete methods, but on single statements. This is different from other evaluations that normally generate multiple mutants for a method and then get a mutation set and calculate the mutation score. As we only have one statement, there are not as many possibilities to mutate the statement, which is why we only generate one mutant. However, this approach fits CbC better, as verification is done for single statements.

Additionally, the selection of applicable mutation operators was performed manually, which is prone to human errors. As we looked at single statements that are not overly complicated, we assessed that this risk is not too high. In either case, we performed sanity checks to make sure that the statement had actually been modified and that it was free of syntactic errors.

External Validity. The main concern of our evaluation is generalizability, as we used only small-sized subject systems. We argue that this is, to the best of our knowledge, the first application of test case generation to a CbC-based process on code level and therefore is a necessary stepping stone to extended analyses in the future. The selected subject systems consist of algorithms and object-oriented projects that represent the complexity of the current implementations in CorC. Another assumption that we made was that verifying the subject systems has to work automatically. In general, there exist only a few open-source projects that fulfill this criterion. Finally, an advantage of having few subject systems is that we can manually inspect the generated test cases and mutations. We also believe that the observed results can be transferred to other languages or techniques with similar specification techniques.

Another threat may be the scalability and performance of our approach. We did not report any performance metrics, as our tool CorC and the evaluation setup are still prototypical and require manual effort to generate the mutants. We rate the benefits and effectiveness of a mutation-based analysis higher than, for example, manually inserting faults or transferring new subject systems to CorC to compare performance and scalability.

RQ2 – Qualitative Evaluation

Internal Validity. With our evaluation setupt, we are unable to evaluate whether the integration of test case generation as a well-known quality assurance technique eases the verification task for non-experts in software verification. To fully prove this hypothesis, it is necessary to collect data about the number of participants that are able to finish a task (with and without help) and what time they need to solve the task with both versions of CorC. We did not collect this data in our evaluation because we focussed on getting suggestions for improvements of the tool support.

External Validity. The design of our qualitative usability refinement study lacks generalizability, as we only had five participants from the same group of people and used small-sized subject systems. The small number of participants allowed us to conduct the experiment and interview individually with each participant. We rate the individual insights that we get in separate interviews higher than a generalizable user study with many participants. Furthermore, the participants all belonged to one group, that is students or former students in computer science. Therefore, they can be ranked as junior software engineers with the necessary background in testing and code understanding. Consequently, our evaluation results are not generalizable to other groups. However, the selected participants belonged to our target group, i.e., software developers that are new to the field of software verification and had the required knowledge of the fundamentals of CorC to participate. The selected participants therefore enabled us to provide evidence that our hypothesis is true, that the integration of testing techniques eases the verification effort in our process, but we are unable to fully prove this hypothesis with our evaluation setup, as discussed above. Using small-sized subject systems for the experiments in the usability refinement study was necessary, as the participants are non-experts in software verification, and therefore the experiments needed to be simple enough for them to solve but to gain experience in using the implemented techniques in CorC to report their opinions. Therefore, we used small algorithms where the implementation was familiar.

4.6. Related Work

In this section, we discuss related work on alternative processes that integrate testing and verification, test case generation in other refinement-based software development approaches, and the evaluation methodologies that we used. Related work for test case generation has already been discussed in Section 4.3.3.

Other Development Processes Integrating Testing and Verification. Ahrendt et al. (2022) describe their vision for a development process that supports the simultaneous and agile development of the three artifacts – specification, tests, and implementation – using a machine learning-based co-pilot. All artifact types shall be checked for conformance and modifications. To ensure conformance, specification and implementation are checked by verification, specification and tests are checked by generating test cases from the specification, and tests and implementation are checked by executing the generated tests. When one artifact of any kind is modified, the other two are adapted accordingly. Our approach in this chapter has the same kinds of artifacts, but ordered into three levels of correctness guarantees. As we support the development with CbC, we can also iteratively pass all three correctness levels for each refinement step, similar to the agile process that Ahrendt et al. (2022) describe. However, the process of Ahrendt et al. (2022) is still a vision and does not concretely describe the techniques they want to use. Furthermore, their focus is to integrate generative AI in the form of a co-pilot for the developer. We discussed how LLM-based techniques can be integrated into the workflow of CorC in (Kodetzki et al., 2025a).

Verification-based test data generation uses the idea of generating test cases from attempts to verify systems with formal specifications (Engel and Hähnle, 2007). Symbolic execution is used to generate test data from path condition formulas. This works well for code with simple branching statements, but not as well for code with generalized loops or recursion, because only a limited number of loop iterations and only a limited recursion depth can be dealt with. Verification-based test data generation has been implemented in the deductive verifier KeY (Ahrendt et al., 2016b) and in Kiasan/KUnit (Deng et al., 2007). A uniform framework for verification and testing has been formalized in HOL/Isabelle for a small target language (Brucker and Wolff, 2009).

There are also multiple approaches that combine testing with other techniques than symbolic execution. For example, Christakis et al. (2012) propose an approach where implicit assumptions of static checkers are made explicit, and unit test cases that test the parts of the program are derived that could not be verified, including the new explicit assumptions. Czech et al. (2015) propose an approach based on conditional model checking to generate test cases out of partially verified programs implemented in the software analysis tool CPAChecker (Beyer and Keremoglu, 2011) and use the concolic testing tool KLEE (Cadar and Nowack, 2021) for the dynamic analysis.

Apart from the development process described by Ahrendt et al. (2022), all other development processes use verification techniques either to generate input data for test cases or after verification to cover parts of the program that could not be verified. The idea of our approach is to test the program first to locate major faults and afterwards verify the program to eliminate faults in corner cases. None of the mentioned approaches has been assessed in a qualitative and quantitative evaluation as we have done.

Other Refinement-based Approaches. Besides the CbC-based program construction approach proposed by Dijkstra (1976) and Kourie and Watson (2012), which we pursue in this work, there are other *refinement-based approaches* that guarantee the correctness of a program under development. We already discussed their relation to CbC as we pursue it in Section 1.1. In the following, we only briefly mention differences and focus on the integration of test case generation in these refinement-based approaches. In the Event-B framework (Abrial, 2010), automata-based system descriptions are refined to concrete implementations. The test generation approaches for Event-B have a different focus than our approach, i.e., using model-based test generation approaches and targeting system-level test cases, such as scenario-based testing and robustness (Savary et al., 2015; Dinca et al., 2011). Morgan (1990) and Back (1998) proposed related CbC approaches. Morgan's refinement calculus is implemented in the tool ArcAngel (Oliveira et al., 2003) and (Back, 2009; Back et al., 2007) developed the tool SOCOS. Neither for ArcAngel nor for SOCOS, there exist test case generation in the form of unit tests, as we propose in this chapter.

Evaluation. In the related works mentioned in the previous paragraphs, the authors only performed quantitative evaluations, if any. For example, (Deng et al., 2007) generate tests for Java library methods and focus on the size and coverage of their generated test suites to determine effectiveness. Czech et al. (2015) evaluate the performance of their integration of testing and verification in comparison to verification without testing. In the quantitative part of our evaluation, we mutated the code of correct subject systems, which mimics typical mistakes that a developer introduces (DeMillo et al., 1978; Jia and Harman, 2011; Ammann and Offutt, 2017). Afterwards, we assess in which of the correctness levels the faulty mutant in the code is detected. The methodology for this evaluation is inspired by mutation testing (Ammann and Offutt, 2017) where the effectiveness of the test suite is evaluated by mutating the subject under test and assessing whether a test suite detects the introduced mutant. Similar to mutation testing, mutation-based analyses have already been used in many other fields to introduce failures for the evaluation (Andrews et al., 2006; Birkemeyer et al., 2022; Knüppel et al., 2021). For example, Knüppel et al. (2021) used mutation-based analysis to assess the quality of specifications. In our experiments, specifications were strong enough except for one case (see Section 4.5.2).

Besides the quantitative part of our evaluation, we also performed a qualitative usability refinement study, since our goal was to generate suggestions for improvements on the current usability of the (still prototypical) tool support in each level of correctness. Qualitative evaluations have not been done for any of the mentioned approaches. For CorC, there are already two user studies (Runge et al., 2020; Runge et al., 2021), that gathered valuable insights into the strengths and weaknesses of CbC as a methodology implemented in CorC as well as feedback from the participants who have used CorC in the experiments of the studies.

4.7. Chapter Summary

Software testing is state-of-the-art for ensuring the quality of a software system, although formal methods, such as CbC, provide stronger guarantees of functional correctness. We believe that two reasons for this phenomenom are that 1) software verification experts are rare in practice, and 2) there is a usability gap between testing tools and formal verification tools. As a result, software verification is rarely performed, even though a system may be safety-critical. The goal of this chapter was to investigate how we can lower the entry barrier to using CbC for non-experts in software verification, with the goal of broadening the user group of CbC. In particular, we investigated whether integrating software testing as a well-known quality assurance technique into the development process of CbC can lower this entry barrier.

In this chapter, we introduced three levels of correctness guarantees into the CbC development process and the tool CorC to ease the verification task for less experienced developers in the field of formal software verification. The first correctness level, *specified*, is supported by improved error messages to detect syntax errors. The second correctness level, *tested*, is new for CbC-based development and is supported by test case generation from the specification of the first correctness level. Test cases show major errors in the program and provide more readable feedback to developers. The third correctness level, *verified*, is supported by counterexample generation. Counterexamples help to detect errors more easily than examining open proof goals. We evaluated our three-level process as implemented in CorC quantitatively and qualitatively. In the quantitative evaluation, we found that almost three-quarters of defects introduced by mutating the source code could already be located at the correctness level *tested*, which is presumably easier to understand for non-experts in software verification. In the qualitative evaluation, we performed a usability refinement study, which resulted in a list of helpful suggestions for improvement of the tool support. The participants rated the conceptual extensions at each correctness level as useful. In conclusion, we have shown that successive levels of correctness guarantees do indeed support the development of verified programs using CbC for non-experts in software verification. For future work, we plan to integrate the feedback from the usability refinement study, extend the evaluation by using larger subject systems, and integrate generative AI as a new technique to further improve usability of CbC.

5. Correct-by-Construction Software Product Lines

The content of this chapter is largely based on Bordis et al. (2020a), Bordis et al. (2020b), Bordis et al. (2022b), and Bordis et al. (2023a). The concept of variable predicates has been written solely for this thesis.

Software product line engineering (Pohl et al., 2005; Apel et al., 2013a) has integrated itself successfully from academia to industry, where it is used to meet the need for custom-tailored software. The primary objective of software product line engineering is to provide efficient techniques to implement a whole product family rather than single software products. Software product line engineering has already been applied in several different sectors, as, for example, in the automotive, avionic, or medical sector (Weiss et al., 2006). One commonality that software systems in these sectors share is their safety relevance, which requires guarantees for behavioral correctness of the software system that exceed standard software testing as a software quality assurance technique. In aviation, this is even required by the corresponding standard (RTCA DO-178C, 2011).

While most verification approaches rely on post-hoc verification (Bruns et al., 2011a; Hähnle and Schaefer, 2012; Thüm et al., 2019), where a program is verified after its implementation, CbC's incremental approach may be beneficial, especially when specifying highly variable software, as the variable implementation is created *incrementally* based on its specification. In contrast, with post-hoc verification, the developer has to specify and verify all variable parts *after* finishing the whole implementation, which becomes even more difficult with growing variability. However, CbC does not contain any notion of variability, neither in the refinement rules nor in the specifications to construct correct-by-construction software product lines (Challenge 4).

In this chapter, we extend CbC to develop software product lines using *feature-oriented programming* (Prehofer, 1997; Apel et al., 2013a) as a product line realization technique. In feature-oriented programming, features are implemented by feature modules that contain the implementation artifacts belonging to a certain feature. Similar to inheritance in object-oriented languages, in a feature module, methods can be added or overridden[1] in a specific order defined by the feature model. Overridden methods can use the `original` keyword to call the previous implementation of that method. The feature configuration then defines the explicit relationship between feature modules. Besides `original` calls, method calls are also variational, since the behavior of a called method also differs for different product variants.

Generally, there are three aspects that need to be adjusted to deal with the variability of software product lines using CbC, as CbC has originally been designed to develop single methods only. The first aspect is the specification of methods to be developed. The second one is refinement rules introducing variability (i.e., `original` and variational method calls). The third aspect is a mechanism to guarantee the side conditions of the refinement rules for all product variants of the product line. In this chapter, we address Research Question 3: *How can we extend Correctness-by-Construction to support the development of highly configurable software systems?* by proposing concepts for all three aspects to specify, implement, and verify variable code structures that are part of feature-oriented programming as a software product line realization technique.

The first aspect that needs to be adjusted to develop software product lines with CbC is the specification formalism. Existing work on specifying software product lines uses *feature-based specifications* (Thüm et al., 2019; Hähnle et al., 2013; Damiani et al., 2012; Agostinho et al., 2008). Usually, the specifications are defined in first-order logic and provided for every method in a feature module, i.e., the specification for each method is given per feature. We introduce two *original predicates* as a feature-based specification technique for CbC. The original predicates mimic the behavior of an `original` call on code level and fit the fine-grained specification style of CbC.

However, using feature-based specifications on its own lacks the capability to specify the behavior of a subset of the product family that exceeds one feature. Specifications for a subset of the product line can be defined using *family-based specifications*. We propose *variable predicates* based on first-order logic predicates that can either evaluate to `true` or `false` (e.g., a predicate `isSorted(array a)` evaluates to `true` if the elements of array `a` are sorted and `false` otherwise). In contrast to regular predicates, the condition under which variable predicates evaluate to `true` or `false` is connected to a presence condition on the features of a product line. This results in varying definitions of the predicate for different product variants and therefore can be applied to define differing behavior of a method for subsets of the product family. Besides adding precision through explicitly specifying properties among a single feature, variable predicates also improve the readability of the specification since they abstract certain parts of a specification while still having descriptive names

[1] Overriding is the technical term in object-orientation when an inherited method implementation is redefined. In feature-oriented programming, the term *method refinement* is used when a feature redefines the implementation of a method that has been introduced by another feature. To distinguish between CbC refinements and feature module refinement steps, we use overriding when we refer to feature module refinements.

of their purpose. The effect on readability is similar to encapsulating certain code in functions, i.e., a standard practice in every programming language to achieve understandable and maintainable code (Martin, 2009).

The second aspect where we extend CbC is the introduction of *CbC refinement rules for variation points* in the code, such as the `original` call and *variational method calls*. The third aspect is directly connected to the new refinement rules, as we need verification strategies to guarantee the correctness of the side conditions of the refinement rules for all product variants in the product line. We propose two different verification strategies for checking the side conditions: *an optimized product-based verification strategy* and a *family-based verification strategy*. Product-based verification strategies usually generate all valid products from the product line and check their correctness in isolation (Thüm et al., 2014a). We optimize this by only checking the *relevant feature sequences* instead of the whole set of valid feature sequences forming valid products. The set of relevant feature sequences is formed by analyzing which features of the product line have an impact on the refinement step that is verified and leaving out other features that lead to redundant side conditions. Still, product-based verification strategies have been shown to be inefficient for post-hoc verification compared to family-based verification strategies (Thüm et al., 2014a; Thüm et al., 2012). Therefore, we propose a family-based verification strategy as an alternative to the product-based verification strategy. Family-based verification strategies construct a metaproduct that simulates the behavior of all possible products in the product line by encoding the variability in the implementation and the specification (Thüm et al., 2014a). For generating the metaproduct in our family-based verification strategy, we transform the CbC method refinements from the feature modules into one CbC method for the family. We introduce feature flags and selection refinements to differentiate between the implementations of different feature modules.

We combine the aforementioned concepts in two phases to develop correct-by-construction software product lines. First, in the *feature-based CbC phase*, the variational specification (original and variable predicates) and the variational refinement rules (`original` call and variational method call) are applied to construct CbC method refinements in feature modules. Naturally, variation points in a product line depend on the implementation and specification of other features in the product line. Therefore, side conditions cannot be checked immediately but require relevant parts of the product line to already be constructed and checked. This is done in the second phase, the *verification phase*, where either the product-based or the family-based verification strategy is applied to check the side conditions.

We implement all concepts (the original predicates and variable predicates for the specification, the variational refinement rules, and the product-based and family-based verification strategy for the side conditions of the refinement rules) in an extension of CorC that we call VarCorC. In the evaluation, we use VarCorC to show the feasibility of our concepts. Furthermore, we compare our concepts to develop correct-by-construction software product lines with post-hoc verification in terms of specification effort and proof efficiency. Last, we compare the product-based with the family-based verification strategy for the side conditions of the refinement rules in terms of proof efficiency to assess which one is more efficient.

This chapter is structured as follows. In Section 5.1, we give an overview of the process to develop correct-by-construction product lines as a whole and motivate each concept with an example. In Section 5.2, we introduce *variational CbC*, which is a feature-based phase where the specification concepts of the original predicates and variable predicates (Section 5.2.1), as well as the new CbC refinement rules (Section 5.2.2), are introduced. The original predicates were first introduced in (Bordis et al., 2022b). In Section 5.3, we propose two verification strategies that can be used to check the side conditions of the refinement rules that we introduce in Section 5.2.2. The product-based verification strategy was first proposed in (Bordis et al., 2020a) to enable the development of single variational methods and extended in (Bordis et al., 2020b) to create whole product lines. The family-based verification strategy was proposed in (Bordis et al., 2022b). All formal definitions in this chapter are based on the definitions of software product lines that we defined in Section 2.3. In Section 5.4, we introduce our tool VarCorC implementing CbC for software product lines with object-orientation that is based on (Bordis et al., 2023b). In Section 5.5, we present the evaluation results, and in Section 5.6, we position our work in the field of software product line specification and verification techniques. Finally, we conclude this chapter in Section 5.7.

5.1. Motivating Example

In this section, we demonstrate our concept of using CbC for software product line development on the *IntegerList* product line,[2] which we introduced in Section 2.3. The *IntegerList* provides basic functionality for maintaining an array of integers. It has five features, and the feature model is shown in Figure 2.3. Even for this rather small example, there are six valid configurations in total (FM = {[*Base*], [*Base*, *Limited*], [*Base*, *Sorted*, *Increasing*], [*Base*, *Sorted*, *Decreasing*], [*Base*, *Limited*, *Sorted*, *Increasing*], [*Base*, *Limited*, *Sorted*, *Decreasing*]}). The number of possible configurations grows exponentially with the number of optional features. Using classical CbC, as introduced in Section 2.2, the developer would have to implement each method six times, i.e., once for each product variant. As mentioned in Section 2.3, we use feature-oriented programming (Prehofer, 1997; Apel et al., 2013a), which provides the `original` call as a code reuse mechanism. The general concept of this chapter does not depend on a particular programming language, as long as the programming language provides the constructs of the Guarded Command Language (Dijkstra, 1975) and method calls. For the concrete statements, we use Java. In this example, we abstract from the use of object-orientation. In Section 5.4, we discuss how we support

[2] The full subject system with all refinement steps and complete specifications is available at https://github.com/KIT-TVA/CorC

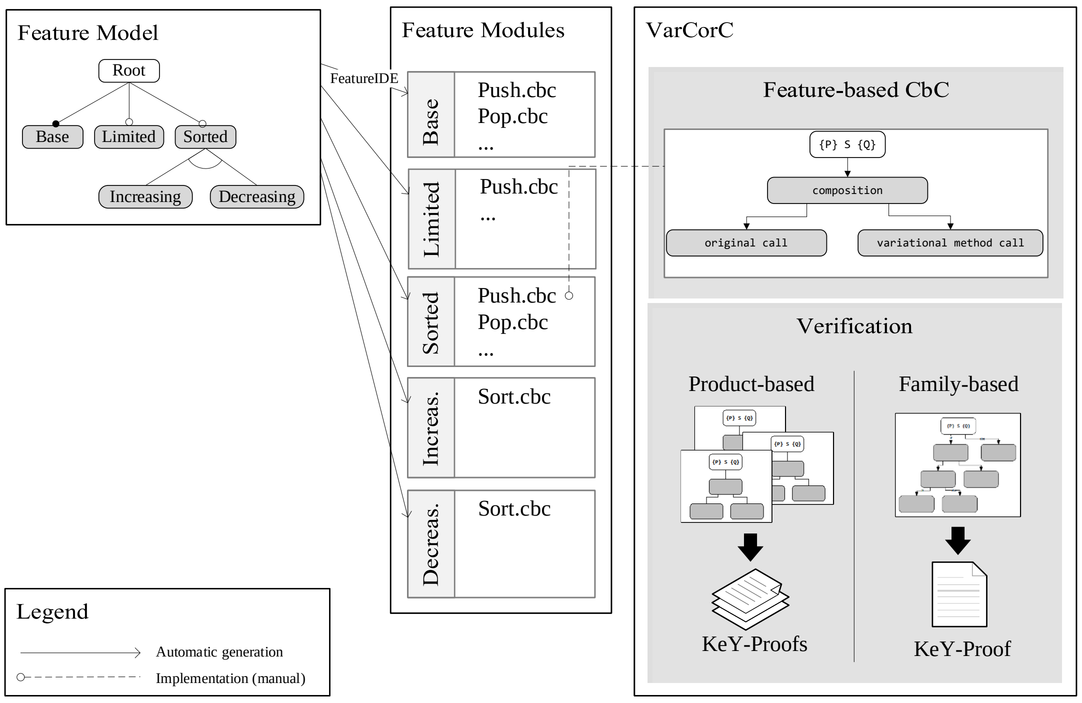


Figure 5.1.: Overview of the correct-by-construction software product line development process (Bordis et al., 2022b). On the left, there is the feature model. For each feature in the feature model, there is a module that contains its CbC implementations. The CbC implementations are separated into a feature-based CbC construction phase and the verification of the applied refinement steps. To verify the refinement steps in CbC implementations, we propose a product-based and a family-based verification strategy.

CbC development for object-oriented software product lines as proposed in (Bordis et al., 2023a). The object model is based on our contribution in Chapter 3.

In Figure 5.1, we give an overview of the CbC development process for software product lines at the example of the *IntegerList* product line. In the upper left corner, there is the feature model of the *IntegerList* product line. For each concrete feature in the feature model, a feature module is created. In these feature modules, the user can implement the features using the tool VarCorC, which supports the concepts presented in this chapter. The CbC files in the feature modules correspond to a method implementation in a feature. The methods `push` and `sort` are implemented in more than one feature module. They override each other in the feature composition order given by the feature model FM. Our overall approach to develop CbC software product lines is divided into two phases: First, the *feature-based CbC phase* includes concepts for *variational specifications* and the extension of the set of refinement rules with two new variational refinement rules, and second, the *verification phase* for checking the correctness of the side conditions of the new refinement rules, which can be performed either with a *product-based* or *family-based* verification strategy. While the feature-based CbC phase has to be done manually by the developer, the verification phase is automated by our tool VarCorC.

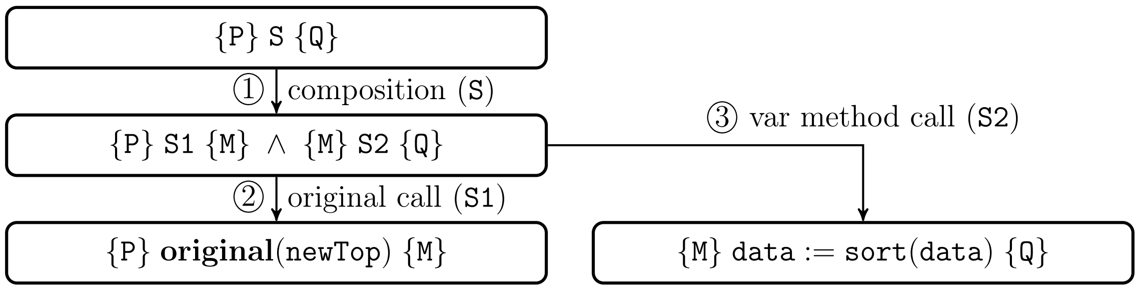


Figure 5.2.: Refinement steps of method push in feature *Sorted*, adapted from Bordis et al. (2020a).

```
P = \original_pre
M = \original_post
Q = \original_post & isSorted(data)
```

Listing 5.1: Variational specification for method push in feature *Sorted* (Bordis et al., 2022b).

5.1.1. Variational Specifications

In the specification, we generally want to formally describe the behavior of a method. This can be difficult in product lines because original and method calls can refer to different implementations, and thus the behavior of a method can vary depending on a product variant. Therefore, we propose a variational specification that abstracts from the varying behavior of original calls and method calls. To achieve this, we extend the specification language with two concepts. First, we propose the original predicates \original_pre and \original_post, which mimic the behavior of original calls in code (see Section 2.3) by referring to the pre- or postcondition of the method implementations that we are currently extending. Second, we introduce *variable predicates*, which have different definitions tied to presence conditions over the features in the feature model.

Original Predicates

In Figure 5.2 and Listing 5.1, we show method push in feature module *Sorted* to illustrate the feature-based development of a method using CbC with variational specifications. In Figure 5.2, we show the implementation of the method using CbC refinement rules. Feature *Sorted* ensures that array data in the *IntegerList* is always sorted. This requirement is implemented in method push by first using the composition refinement rule to split the abstract statement into two ①. The first abstract statement then uses an original call to reuse the implementation of the push method in a smaller feature according to the feature composition order, which adds the new element ②. The second abstract statement calls a method sort, which sorts the array ③. In Listing 5.1, we show the corresponding specifications of the CbC refinement tree of method push for precondition P, intermediate condition M, and postcondition Q to form the variational specification. In CbC development, we always

```
isSorted(int[] a) = (\forall int i; 0 <= i & i < a.length − 1; a[i] <= a[i + 1]) |
                    (\forall int i; 0 <= i & i < a.length − 1; a[i] >= a[i + 1])
```

Listing 5.2: Definition of predicate `isSorted` without variability.

$$\texttt{isSorted(int[] a)} = \begin{cases} \texttt{(\textbackslash forall int i; 0 <= i \& i < a.length} - 1; & \\ \texttt{a[i] <= a[i + 1])} & \textit{iff} \quad \textit{Sorted} \wedge \textit{Increasing} \\ \texttt{(\textbackslash forall int i; 0 <= i \& i < a.length} - 1; & \\ \texttt{a[i] >= a[i + 1])} & \textit{iff} \quad \textit{Sorted} \wedge \textit{Decreasing} \end{cases}$$

Listing 5.3: Variable predicate definition of `isSorted`.

start with a pre- and postcondition pair. For our example, we can set the precondition `P` to `\original_pre`, since we just want to take the precondition from the previous implementation of method `push` and do not need to add anything special in this case. For the postcondition `Q`, we want to express the newly implemented behavior of `push`, which is that the array is now also sorted. We use the predicate `\original_post` to express the part with the new element in the postcondition, and the (variable) predicate `isSorted(int[] a)` for the sorting property. The `\original_pre` and `\original_post` specifications are then replaced, depending on the feature configuration, either by the pre-/postcondition of feature *Base*, which specifies the behavior that the new element is added regardless of the array length, or by those defined in feature *Limited*, including length limit.

In previous work (Bordis et al., 2020a), we also used an `original` keyword in the specification to refer to previous specifications of methods, which was first proposed for post-hoc verification (Thüm et al., 2019). In that work, preconditions are only allowed to refer to preconditions, and postconditions are only allowed to refer to postconditions. This concept works well for post-hoc verification because there is usually only one pre- and postcondition per method. Using CbC, we also need to specify intermediate conditions for the composition refinement rule, which are the postcondition and the precondition for the two new abstract statements. Therefore, we adapt the specification mechanism such that instead of using one `original` keyword in specifications that implicitly refers to either the pre- or postcondition, there are now two original predicates (`\original_pre` and `\original_post`) that explicitly refer to either the precondition or the postcondition of the next smaller feature in a feature configuration. As a result, the original predicates can also be used in intermediate conditions. In our example, we set the intermediate condition `M` to `\original_post`, as this exactly captures the behavior of our method at this point (we only called `original` in the code yet).

Variable Predicates

In Listing 5.2, we show the definition of predicate `isSorted` without using variability. Since the *IntegerList* defines two subfeatures *Increasing* and *Decreasing* for feature

Sorted, the predicate must specify both behaviors, *increasing* sorting and *decreasing* sorting, since it depends on the selected feature configuration, which behavior is executed. Therefore, both cases are connected with a disjunction in the definition of predicate `isSorted`. Even though this specification describes the behavior correctly for each product variant, it is not precise because it does not specify which product variants of the software product line sort the array in increasing and which in decreasing order, only that one of these orders is satisfied. Imagine adding a feature that implements functionality for search algorithms (e.g., finding the maximum element) on this array. These can be efficiently implemented on sorted arrays, but the order in which the elements are sorted must be known.

With variable predicates, it is possible to define this sorting criterion in a family-based way, such that only product variants containing features *Sorted* and *Increasing* use the condition that the array is sorted in an increasing order and product variants containing features *Sorted* and *Decreasing* use the condition that the array is sorted in a decreasing order. In Listing 5.3, we show the alternating definitions of the variable predicate `isSorted` that are coupled to a presence condition on the feature model. Compared to the definition without variability in Listing 5.2, the definition using this variable predicate is more precise, while still concise and easy to understand because the concrete definition is abstracted and modularized.

5.1.2. Product-based Verification

As illustrated in Figure 5.1, we propose two different strategies to prove the side conditions of the CbC refinement rules in product lines. For both, we need to guarantee their correctness for all valid product variants of the product line by resolving the `original` calls in the code and the original predicates, as well as variable predicates in the specifications. The simplest product-based verification strategy is to generate all valid product variants from the feature model and check the correctness of each product variant accordingly. Since the number of product variants increases exponentially with the number of optional features, we propose an optimized product-based verification strategy where we only generate a set of *relevant feature sequences* that can be used to compose the variants of a method that need to be checked. Depending on which constructs are used in the specification and code (original predicates, variable predicates, original call refinement rule, and variational method call refinement rule), this set is different. In the following, we describe how the relevant feature sequences are determined and checked in our example for the original call refinement step ② with original predicates in the specification, and the variational method call refinement step ③ with original predicates and variable predicate in the specification.

In the product-based verification strategy, we compute the relevant feature sequences for the `original` calls whenever the developer wants to check the side conditions of a CbC refinement containing one of them. For the `original` call and the original predicates, we use the feature model and the information about which feature modules implement the method. To guarantee the correctness of the refinement step ②, the

```
P_Limited  = \original_pre
Q_Limited  = (data^old.length < LIMIT) → \original_post
```

```
P_Base  = data != null
Q_Base  = contains(data, newTop) & containsAll(data^old, data)
```

Listing 5.4: Contract of method `push` in features *Limited* and *Base* (Bordis et al., 2022b). For brevity, we left out the case when the length of `data` is greater than or equal to `LIMIT` in the contract for feature *Limited*.

```
{P1} push$$Base(newTop) {M1} with                          Base • Sorted
P1 = data != null
M1 = (contains(data, newTop) & containsAll(data, data^old))
```

```
{P2} push$$LimitedBase(newTop) {M2} with          Base • Limited • Sorted
P2 = data != null
M2 = ((data^old.length < LIMIT) → (contains(data, newTop) & containsAll(data, data^old)))
```

Listing 5.5: Composed specifications for `original` call in method `push` in feature *Sorted* (Bordis et al., 2022b).

`original` call can either be replaced by the implementation of `push` in feature *Base* or by a composition of the features *Base* and *Limited*. So, for every side condition containing an `original` call or an original predicate, two explicit versions of this side condition with replaced original predicates are generated and checked. Given the contracts of method `push` in features *Limited* and *Base* in Listing 5.4, the two composed specifications shown in Listing 5.5 have to be considered to guarantee the correctness of refinement step ② in our example. The replacement of `\original_pre` and `\original_post` in the specification for a concrete feature configuration is done by *contract composition*. The `original` call has been replaced by a concrete method call with the composed implementation of `push` for both configurations that have to be considered (denoted as `push$$Base`/`push$$LimitedBase`).

As seen in this example, there can be several replacements for an `original` call. Consequently, we need to define a *set of relevant feature sequences* for an `original` call that correctly captures all these method calls needed to guarantee the correctness of an `original` call. This set is defined by the particular context that regulates the use of the `original` call. Afterwards, we propose an optimized product-based verification strategy to check the side conditions of the `original` call refinement rule for CbC that uses this set of relevant replacements to ensure the correctness of the product line for every possible product variant.

Next, we want to guarantee the correctness of the method call refinement ③ where method `sort` is called (see Figure 5.2). As seen in Figure 5.1, this method is defined in two feature modules, namely *Increasing* and *Decreasing*. As the implementation of method `sort` changes according to the selected feature configuration, we refer to this call of method `sort` as a *variational method call*. Similar to the `original` call,

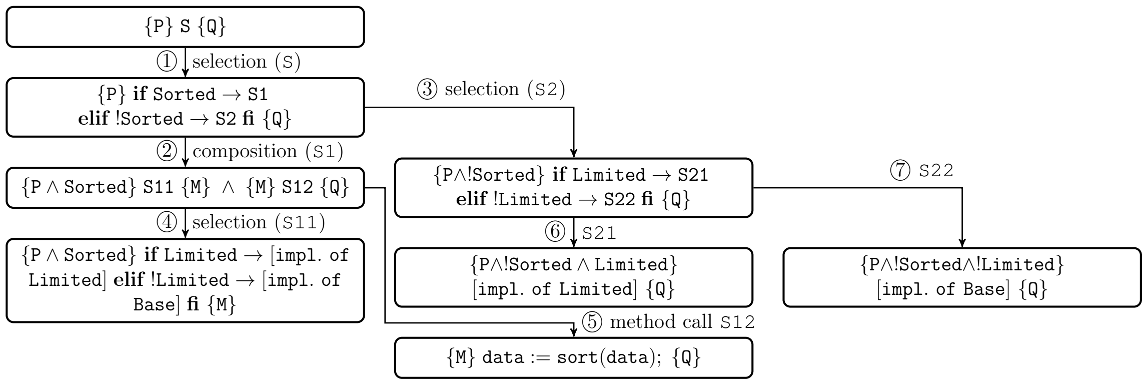

{P} S {Q}
① selection (S)
{P} if Sorted → S1 elif !Sorted → S2 fi {Q}
③ selection (S2)
② composition (S1)
{P ∧ Sorted} S11 {M} ∧ {M} S12 {Q}
{P∧!Sorted} if Limited → S21 elif !Limited → S22 fi {Q}
⑦ S22
④ selection (S11)
⑥ S21
{P ∧ Sorted} if Limited → [impl. of Limited] elif !Limited → [impl. of Base] fi {M}
{P∧!Sorted ∧ Limited} [impl. of Limited] {Q}
{P∧!Sorted∧!Limited} [impl. of Base] {Q}
⑤ method call S12
{M} data := sort(data); {Q}

Figure 5.3.: Metaproduct of method push (Bordis et al., 2022b).

we have to establish side conditions for all possible contracts of sort to guarantee correctness. In our example, method sort either refers to the implementation in feature *Increasing* or *Decreasing* where method sort has different contracts, and both contracts need to be checked when method sort is called. As a result, we need to define a new refinement rule for variational method calls, and for the product-based verification strategy, we need to define the corresponding set of relevant replacements, which differs from the set for an original call as we have seen in this example.

To guarantee the correctness of this refinement step, also the original predicates and the variable predicate isSorted in the specification need to be replaced by a concrete definition that can be evaluated. The composition of the original predicates uses the same set of relevant feature sequences as described for refinement step ②. When this is combined with the set of relevant feature configurations of the variational method call, we get the four feature sequences 1) [*Base, Sorted, Increasing*], 2) [*Base, Limited, Sorted, Increasing*], 3) [*Base, Sorted, Decreasing*], and 4) [*Base, Limited, Sorted, Decreasing*]. The variable predicate is defined for two presence conditions (see Listing 5.3): a) $Sorted \wedge Increasing$ and b) $Sorted \wedge Decreasing$. These can be evaluated on the set of relevant feature sequences and predicate isSorted can be replaced with definition a) for the relevant feature sequences 1) and 2) and with definition b) for the relevant feature sequences 3) and 4). The complete example is given later in Section 5.3.1.

5.1.3. Family-based Verification

Family-based verification strategies aim to express the behavior of all products in a single metaproduct (Thüm et al., 2014a). We generate one metaproduct for each CbC method that simulates the behavior of all variants of that method in the product line. We introduce feature flags representing the selection of a feature from the feature model. Using these feature flags, we translate the compile-time variability of the original calls into runtime variability using the selection refinement rule to distinguish between the implementations of different features. Similarly, we can create a metaspecification using the feature flags that yield implications of the form

```
P_B = (Base → data != null)
P_L= (Limited → \original_pre) & (!Limited → P_B)
= (Limited → P_B) & (!Limited → P_B)
P_S= (Sorted → \original_pre) & (!Sorted → P_L)
= (Sorted → P_L) & (!Sorted → P_L)
P = (Sorted → ((Limited → (Base → data != null)) & (!Limited → (
    Base → data != null)))) & (!Sorted → ((Limited → (Base → data !=
     null)) & (!Limited → (Base → data != null))))

Q_B = (Base → (contains(data, newTop) & containsAll(data^old, data)))
Q_L = (Limited → ((data^old.length < LIMIT) → \original_post)) & (!
    Limited → Q_B) = (Limited → ((data^old.length < LIMIT) → Q_B)) & (!
    Limited → Q_B)
Q_S = (Sorted → (\original_post & isSorted(data))) & (!Sorted → Q_L) =
    (Sorted → (Q_L & isSorted(data))) & (!Sorted → Q_L)
Q = (Sorted → ((Limited → (data^old.length < LIMIT) → (Base → (
    contains(data, newTop) & containsAll(data^old, data)))) & (!Limited
    → (Base → (contains(data, newTop) & containsAll(data^old, data)))) &
     (Sorted & Increasing → (\forall int i; 0 <= i & i <
    data.length-1; data[i] <= data[i+1])) & (Sorted & Decreasing →
    (\forall int i; 0 <= i & i < data.length-1; data[i] >= data[i+1])))
    ) & (!Sorted → (Limited → (data^old.length < LIMIT) → (Base → (
    contains(data, newTop) & containsAll(data^old, data))) & (!Limited →
     (Base → (contains(data, newTop) & containsAll(data^old, data))))
```

Listing 5.6: Specification of the metaproduct for method `push` (Bordis et al., 2022b). The conditions with subscripts are intermediate conditions for single features. The variable predicate `isSorted` and the resolved condition is colored in teal. The feature flags are colored lilac.

$(f \rightarrow \mathtt{P}_f)$ for a precondition and $(f \rightarrow \mathtt{Q}_f)$ for a postcondition with feature f and the corresponding specifications $\mathtt{P}_f$ and $\mathtt{Q}_f$.

In Figure 5.3, we show an excerpt of the metaproduct of method `push` of the *IntegerList* product line. Method `push` is implemented by features *Base*, *Limited*, and *Sorted*, where *Sorted* is the largest and *Base* the smallest feature according to the feature composition order. Starting with the largest feature according to the feature composition order, a selection refinement rule ① is applied to distinguish between the cases where feature *Sorted* is selected and cases where it is not. For the case where it is selected, we can add the corresponding implementation for `push` from Figure 5.2 (②, ④, and ⑤). Thereby, the `original` call (Figure 5.2 step ②) has already been resolved. Otherwise, we continue with the next smaller feature, which is *Limited*, and add a selection statement that evaluates if *Limited* is selected or not ③. For the case that feature *Limited* is selected, we have the refinement steps of method `push` in feature *Limited* ⑥. If it is not selected, the refinement steps of feature *Base* are inserted ⑦. In Figure 5.3, we abbreviate the insertion of refinement steps of feature *Limited* and *Base* with the notation [`impl. of Limited/Base`].

Besides the encoding of the implementation, also the specification must be encoded in the metaproduct. For the metaproduct of method `push` in Figure 5.3, we give the corresponding specification in Listing 5.6 using the variational specifications given in Listings 5.1 and 5.4. The composed specifications that are used in the metaproduct are called precondition `P` and postcondition `Q`. The specifications with the first letters of the features as subscripts represent intermediate conditions for better understanding. For the features *Limited* and *Sorted*, we add implications for when the respective feature is selected or deselected (see $\mathtt{P}_{Sorted}/\mathtt{Q}_{Sorted}$ and $\mathtt{P}_{Limited}/\mathtt{Q}_{Limited}$ in Listing 5.6). The predicates `\original_pre` and `\original_post` are resolved by replacing them with the corresponding specification of the previous feature. For example, in $\mathtt{Q}_{Sorted}$ `\original_post` is replaced by $\mathtt{Q}_{Limited}$, since *Limited* is the next smaller feature that implements method `push`. The composed postcondition `Q` starts with $\mathtt{Q}_{Sorted}$ and resolves any `\original_pre` and `\original_post` with $\mathtt{P}_{Limited}/\mathtt{Q}_{Limited}$ and $\mathtt{P}_{Base}/\mathtt{Q}_{Base}$ respectively. Variable predicates are encoded using the defined presence conditions PC_i and the corresponding definitions $formula_i$ in the form $PC_i \rightarrow C_i$. In Listing 5.6, the encoding of the variable predicate `isSorted` is marked in teal color in postcondition `Q`. With this family-based verification strategy, it is possible to only conduct one proof for the entire method, but still guarantee correctness for each product of the product line when combined with a family-based encoding of the implementation. The encoded specifications can then be used to check the side conditions of the CbC refinement rules of the metaproduct.

5.2. Variational CbC

In this section, we formally introduce *variational CbC* for the feature-based CbC phase that we described in the previous section. Variational CbC consists of a variational specification with original predicates and variable predicates, and refinement rules for `original` calls and variational method calls. Since all these constructs are related to specifications in other feature modules, sometimes checking the side conditions cannot be done immediately. The two verification strategies that we propose to check whether the side conditions of the proposed refinement rules hold are presented in Section 5.3.

5.2.1. Variational Specification

In feature-oriented programming, the developer creates different refinements of a method in different feature modules that can override each other with respect to the feature composition order. These method refinements can use the predicate `original` to call the implementation of the refinements that have been defined before, i.e., in feature modules that belong to features that are smaller according to the feature composition order. Furthermore, we identified that also method calls may have varying implementations. This may completely change method behavior in a way that violates the Liskov principle (Thüm et al., 2019). Thus, it is crucial to be able to include variability in method contracts as well. For example, when a new product variant changes functionality of a method by refining it, the original contract of that method will most likely be insufficient to specify its behavior. In that case, the contract needs to be adapted as well using a specification technique that introduces variability. In the following, we present two concepts that add variability to the contracts. First, we propose original predicates that mimic the behavior of `original` calls in the code, adapted to the fine-grained specifications used in CbC. Second, we propose variable predicates, which help to define precise specifications amongst original calls using presence conditions on the feature model.

Our concept for the original predicates and variable predicates is based on the definition of predicates in first-order logic. In first-order logic, predicates evaluate to a truth value and are defined as a part of the first-order signature (Smullyan, 1968). Syntactically, a predicate is defined with a name, an arity, and the types of these arguments. Semantically, an interpretation for which argument tuples in the domain the predicate evaluates to `true` is defined. We define an n-ary predicate by a first-order logic formula *formula* that can use the arguments of the predicate and bound variables.

Definition 5.1: Predicate
Let predicate $\texttt{p}(\texttt{type arg}_1,\ \dots,\ \texttt{type arg}_\texttt{n})$ be an n-ary predicate with typed arguments $\texttt{arg}_1, \dots, \texttt{arg}_\texttt{n}$ and a first-order logic formula *formula* defined over free variables $\texttt{arg}_1, \dots, \texttt{arg}_\texttt{n}$. Then we can evaluate $\texttt{p}(\texttt{t}_1,\ \dots,\ \texttt{t}_\texttt{n})$ with first-order terms $\texttt{t}_1, \dots, \texttt{t}_\texttt{n}$ of the correct type as follows:

$$\texttt{p}(\texttt{t}_1,\ \dots,\ \texttt{t}_\texttt{n}) \ = \ \mathit{formula}[\texttt{arg}_\texttt{i}/\texttt{t}_\texttt{i}]$$

Original Predicates

The original predicates that we introduce in this section are related to one of the six contract composition techniques that Thüm et al. (2019) proposed for feature-oriented programming and post-hoc verification. In their evaluation, Thüm et al. (2019) found that four of them are superior of the other two because of their applicability and that other mechanisms were able to replicate them. In previous work (Bordis et al., 2020a), we used a contract composition mechanism called *explicit contract refinement*, which has the largest applicability, to compose the contracts (i.e., the pre- and postconditions of the starting triples) of variational methods. For post-hoc verification, which is the context in which Thüm et al. (2019) proposed this mechanism, it is sufficient to only compose the contracts per method, as there are generally only contracts for complete methods. However, CbC is more fine-grained by introducing Hoare triple specifications for each statement. Therefore, in this section, we adapt the definition of explicit contract refinement to allow more fine-grained specification reuse for CbC.

Explicit contract refinement offers a keyword to explicitly refer to the contracts of the original method refinements, similar to the `original` call in feature-oriented programming. In the mechanism that Thüm et al. (2019) proposed, this keyword

```
P = data != null                                          Base • Sorted
Q = (contains(data, newTop) & containsAll(data, data^old)) & (isSorted(
    data))
```

Listing 5.7: Composed contract for method push with $fc = [Base, Sorted]$ (Bordis et al., 2022b).

is called \original and refers to the pre- or postcondition of the refined contract (i.e., preconditions are always composed with preconditions and postconditions always with postconditions). We adapt this mechanism for CbC by using the predicates \original_pre and \original_post instead of just \original to explicitly refer to either the pre- or postcondition of the refined method so that we can also compose specifications of Hoare triples that do not come from the pre- and postcondition from the starting triple (Kuiter, 2020). We forbid to use original predicates if the method refinement belongs to the smallest feature according to the feature composition order, as they could not be resolved. Considering our example in Figure 5.2, we can use \original_post in the intermediate condition M to explicitly refer to the postcondition of the refined method which was not possible before, because the intermediate condition is both, a pre- *and* a postcondition, and therefore it was not uniquely identifiable.

Thüm et al. (2019) define original informally as a keyword. In this chapter, we formally define \original_pre and \original_post as predicates that are defined with a formula that is related to the pre- or postcondition, but whose concrete definition depends on the verification strategy that will be introduced in Section 5.3. We define the original predicates for CbC as follows:

Definition 5.2: Original Predicates \original_pre and \original_post
Let $\mathbb{P}$ be the set of all preconditions of contracts defined in a software product line and $\mathbb{Q}$ be the set of all postconditions of contracts defined in a software product line. Then we define the original predicates \original_pre and \original_post as 0-ary predicates where \original_pre is related to some precondition $\mathtt{P} \in \mathbb{P}$ and \original_post is related to some postcondition $\mathtt{Q} \in \mathbb{Q}$. We can evaluate the original predicates by either using product-based explicit contract composition (Definitions 5.11 and 5.12) or the replacement operator of the family-based encoding (Definition 5.17).

The introduced predicates \original_pre and \original_post remain uninterpreted as long as the information about the concrete feature configuration is missing. Consequently, specifications containing one of these predicates cannot be proven until the method refinements in the smaller features have been specified and appropriately composed for a product-based verification or encoded into a family-based metaspecification. Both, product-based composition and family-based encoding, are explained in Section 5.3.

Example 5.1. *In Section 5.1, we introduced the IntegerList product line and method* `push`*, that is implemented by the features* Base, Limited*, and* Sorted*. The contracts for all features have been shown in Listings 5.1 and 5.4. In Listing 5.7, we show the resulting composed contract for feature configuration $fc = [Base, Sorted]$. In the following, we describe the two alternatives how the original predicates in the postcondition of method* `push` *can be replaced with either contract composition in a product-based way or encoded in a family-based way. The original predicates in the precondition can be replaced analogously.*

The postcondition of method `push` *in feature* Base *expresses that the new element* `newTop` *has been added to array* `data` *and that all elements that have been stored in* `data` *are still contained after the execution of method* `push` *(see Listing 5.4). The first predicate,* `contains(int[] a, int elem)`*, evaluates to* `true` *if array* `a` *contains the integer* `elem` *and to* `false` *otherwise. The second predicate,* `containsAll(int[] a, int[] b)`*, evaluates to* `true` *if array* `b` *contains all elements that are contained in array* `a` *and* `false` *otherwise. Feature* Sorted *refines the* `push` *method such that the array remains sorted (see Section 5.1). As the contract should reflect the behavior of method* `push`*, we add a variable predicate* `isSorted(int[] a)` *to its postcondition if feature* Sorted *has been selected. We define the postcondition of the* `push` *method refinement in feature* Sorted *as* $\mathtt{Q}_{Sorted}$ = `\original_post & isSorted(data)`*. According to Definition 5.2, we can either replace* `original_post` *in a product-based way with* $\mathtt{Q}_{Base}$ *or in a family-based way with* `Base` $\rightarrow$ $\mathtt{Q}_{Base}$*.*

Variable Predicates

While the original predicates are used to describe the behavior of `original` calls in the code, we found that this specification technique alone is not expressive enough to specify the variability in a software product line precisely. Therefore, we introduce variable predicates as a family-based mechanism that can be used to specify the variying behavior of a product line (e.g., resulting through variational method calls) and describe their application rules.

In contrast to standard first-order logic predicates, we define *variable predicates* to have multiple definitions for different product variants of a software product line. Syntactically, variable predicates are defined and used in the same way as first-order logic predicates. However, we adapt the definition of a variable predicate with a set of specification cases (first-order logic formulas $formula_1$, ..., $formula_m$) that are each connected to a presence condition on the feature model (PC_1, ..., PC_m). A presence condition is a propositional logic formula defined over the features of the feature model and states for which feature configurations the corresponding definition of the predicate is valid. A variable predicate is therefore always interpreted with respect to a certain feature configuration.

Definition 5.3: Variable Predicate
Let variable predicate $\mathtt{varP(type\ arg_1,\ ...,\ type\ arg_n)}$ be an n-ary predicate with typed arguments $\mathtt{arg_1}, ..., \mathtt{arg_n}$, first-order logic formulas $formula_1, ..., formula_m$ defined over free variables $\mathtt{arg_1}, ..., \mathtt{arg_n}$, and presence conditions PC_1, ..., PC_m propositional logic formulas over the features defined in the feature model FM. Then we can evaluate $\mathtt{varP(t_1,\ ...,\ t_n)}$ with first-order terms $\mathtt{t_1}, ..., \mathtt{t_n}$ of the correct type for a feature configuration $fc \in$ FM using valuation of a feature configuration v_{fc} from Definition 2.6 as follows:

$$\mathtt{varP(t_1,\ ...,\ t_n)} = \begin{cases} formula_1[\mathtt{arg_i}/\mathtt{t_i}], & \textit{iff}\ \ v_{fc} \vDash PC_1 \\ ...\ , & \\ formula_m[\mathtt{arg_i}/\mathtt{t_i}], & \textit{iff}\ \ v_{fc} \vDash PC_m \end{cases}$$

Example 5.2. *We already introduced the variable predicate* `isSorted` *in Section 5.1 with an increasing and decreasing sorting order. The presence condition for the increasing sorting order is* $Sorted \land Increasing$ *and for the decreasing order it is* $Sorted \land Decreasing$*. To evaluate predicate* `isSorted`*, it is replaced by the definitions of those parts of the variable predicate whose presence conditions are fulfilled by the considered feature configuration. For example, for a feature configuration [Base, Sorted, Increasing] the features are evaluated as follows:* $Base = true$, $Limited = false$, $Sorted = true$, $Increasing = true, Decreasing = false$*. Therefore, the first presence condition* $Sorted \land Increasing$ *evaluates to* $true$ *and the second presence condition* $Sorted \land Decreasing$ *evaluates to* $false$*. Consequently, variable predicate* `isSorted` *is replaced with the formula for the increasing sorting order (see Listing 5.3).*

Corner Cases of the Presence Conditions. In Definition 5.3, we did not restrict the selection of the presence conditions $PC_1, ..., PC_m$ in variable predicates. Consequently, there are two corner cases that have to be considered for the evaluation of a variable predicate: (1) Multiple presence conditions evaluate to `true` and (2) no presence condition evaluates to `true` for a given feature configuration. In case (1), where multiple conditions are fulfilled, we consider the definitions of all fulfilled presence conditions by building their conjunction.

In case that none of the presence conditions is fulfilled for a feature configuration, the variable predicate would evaluate to an undefined value. Adding a default case to the definition where the predicate is either evaluated to `true` or `false` would lead to false assumptions during verification. For example, imagine a postcondition `contains(data, newTop) && isSorted(data)` using the variable predicate `isSorted`, but for our considered feature configuration of a product none of the defined presence conditions evaluates to `true`. If we had a default case where the variable predicate is evaluated to `true` when no presence condition is fulfilled, a deductive verifier would be able to verify a method with this contract even though it does not sort array `data`. This is because the postcondition states that it fulfills variable predicate `isSorted` when in fact it is only evaluated to `true` because of the default case. The same issue occurs for a default evaluation to `false` when using a negation of the predicate instead (`contains(data, newTop) && !isSorted(data)`). Consequently, by accidentally or purposely omitting a definition for certain feature configurations, verification might be successful when in fact it should not be.

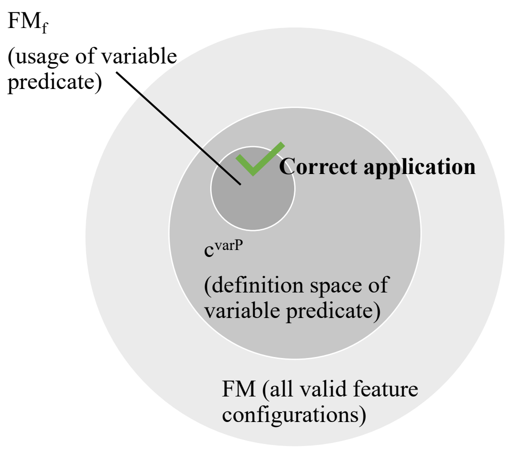
Since it is neither possible to define a default interpretation nor to leave certain cases undefined, we restrict the cases in which a variable predicate can be used instead. Therefore, we define an application rule that defines the cases where a variable predicate is resolvable in any feature configuration that can include that specification. To express this condition logically, we first define the definition space of a variable predicate `varP`, which is the set of valid feature configurations $c^{\texttt{varP}}$ where each feature configurations evaluates to true for at least one of the presence conditions PC_1, ..., PC_m of a variable predicate **varP**(**type arg**$_1$, ..., **type arg**$_n$). Therefore, we first define the set of valid feature configurations that fulfill one presence condition using the valuation of a feature configuration from Definition 2.6 and second, we define the definition space of a variable predicate as the conjunction of the sets of feature configurations for the individual presence conditions of a variable predicate.

Definition 5.4: Feature Configurations for a Presence Condition FM_{PC}
Let $\text{FM}_{PC} = \{fc \in \text{FM} | v_{fc} \vDash PC\}$ be the set of valid feature configurations of feature model FM that evaluate presence condition PC to `true`.

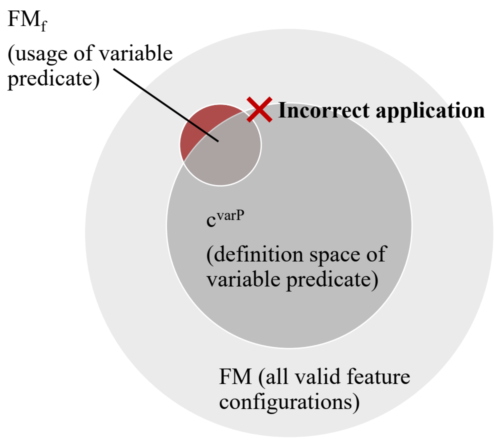
Definition 5.5: Definition Space of a Variable Predicate $c^{\texttt{varP}}$
Let `varP` be a variable predicate with m presence conditions $PC_1, ..., PC_m$. Then we define the definition space of variable predicate `varP` as the conjunction of the sets of feature configurations for each individual presence condition PC_i:

$$c^{\texttt{varP}} = \bigcup_{i=1}^{m} \text{FM}_{PC_i}$$

Besides the presence conditions PC_1, ..., PC_m of a variable predicate, also the code and specifications of a product line are bound to a certain presence condition which states for which feature configuration this code and specification is included. In this chapter, we use the composition-based software product line realization technique feature-oriented programming, where the presence condition is defined indirectly by the feature module that contains this specification (e.g., the specification of Listing 5.1 is defined in feature module *Sorted* and therefore the presence condition of the pre- and postcondition *Sorted*). Since the presence condition in feature-oriented programming is always just one feature, we can use Definition 2.5 for the set of feature configurations


FMf
(usage of variable predicate)
Correct application
cvarP
(definition space of variable predicate)
FM (all valid feature configurations)

(a) Applicable


FMf
(usage of variable predicate)
Incorrect application
cvarP
(definition space of variable predicate)
FM (all valid feature configurations)

(b) Not Applicable

Figure 5.4.: Application rule for a variable predicate. Application is allowed when the specification does not exceed the definition space of the variable predicate.

defined in feature module f where a variable predicate is applied (FM$_{Sorted}$ in our example).

Then variable predicate `varP` is applicable in a specification of feature f if set FM$_f$ of all feature configurations containing feature f is a subset of $c^{\texttt{varP}}$ and therefore fulfills the following applicability condition.

Definition 5.6: Applicability Condition for Variable Predicates
Let `varP` be a variable predicate that is defined over m presence conditions $PC_1, ..., PC_m$ and $f \in FM$ be the feature where `varP` is used in the specification of a Hoare triple `{P} S {Q}`. Then `varP` is only allowed to be used in precondition `P` and postcondition `Q` iff it holds that

$$FM_f \subseteq c^{\texttt{varP}} := \forall fc \in FM_f : fc \in c^{\texttt{varP}}$$

This means that the set of feature configurations that the specification using the variable predicate occurs in (defined by feature f of the feature module) is not allowed to exceed the definition space of that variable predicate (defined by PC_1, ..., PC_m). This ensures that an undefined state never occurs. In Figure 5.4, we show the application rule in a Venn diagram for a variable predicate used in feature f. In Figure 5.4a, FM$_f$ is a subset of $c^{\texttt{varP}}$ and therefore the variable predicate is applicable and in Figure 5.4b FM$_f$ is not a subset of $c^{\texttt{varP}}$ and consequently not applicable.

5.2.2. Refinement Rules for Variation Points

On the code level of a product line, we identified two variation points for feature-oriented programming: `original` calls and variational method calls. To guarantee that these two variation points are correctly implemented during the development with CbC, we need refinement rules that ensure the correctness of all product variants affected by these variation points. Since these variation points depend on different feature modules, we first define the *context of a refinement rule*. This context serves as a basis to differentiate between different implementations of one method in several feature modules and captures the method that is currently refined, the feature this refinement belongs to, and the feature model which defines the dependencies of all features in the product line.

Definition 5.7: Context of a Refinement Rule context(`S`) (Bordis et al., 2022b)
We define the *context* of a refinement rule with abstract statement `S` that is refined as *context(*`S`*)=(*FM*, f,* `m`*)* with feature model FM, feature *f*, and method *m*, which is implemented in the feature module impl(f).

Original Call

In feature-oriented software product lines, methods are implemented across several feature modules to refine a method according to a feature's behavior (Apel et al., 2013a). In feature-oriented programming as defined by FeatureHouse (Apel et al., 2013b; Apel and Lengauer, 2008), an `original` call is used to reuse the implementation of the method in the next smaller feature according to the feature composition order of the feature model. Therefore, the actual method that is called in place of an `original` call depends on the feature configuration. It is forbidden to apply an `original` call refinement rule in the smallest feature according to the feature composition order, as it could not be resolved. During the feature-based CbC phase, an `original` call is introduced but it may not be verifiable immediately. The reason is that the replacement for the `original` call does not only depend on the realization technique (feature-oriented programming in our case), but also on the verification strategy. For example, in a simple product-based verification strategy, all individual product variants are composed according to the software product line realization technique and verified in isolation. The `original` call is therefore directly replaced by a call to the next smaller method refinement according to the feature configuration. When using a family-based verification strategy, a metaproduct of the method is created using runtime variability. Using a family-based verification strategy, an `original` call is replaced with a selection that introduces the different cases that the `original` call can be replaced with for all valid feature configurations. Therefore, we define the original call refinement rule in the feature-based CbC phase with both verification strategies as options in the side condition.

Definition 5.8: Original Call Refinement Rule
Let the context of the abstract statement `S` be $context(\mathtt{S}) = (\mathrm{FM}, f, \mathtt{m})$. Then $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ *can be refined to* $\{\mathtt{P}\}\ \mathtt{b} := \mathtt{original}(a_1, ..., a_n)\ \{\mathtt{Q}\}$ iff one of the verifications strategies: *product-based composition* (Definition 5.14) or *family-based encoding* (Definition 5.19) is applied for method `m` and the respective side conditions hold.

The idea for this original call refinement rule during the feature-based CbC phase is to define a refinement rule that allows to insert the `original` call as programming construct and abstracts from the concrete verification strategy in the side condition. In this chapter, we propose a product-based verification strategy in Section 5.3.1 and a family-based verification strategy in Section 5.3.2 that concretize the side condition that we defined here. But theoretically, more strategies can be added to this rule. For example, in related work (Kodetzki et al., 2024), we experimented with a strategy that creates partial proofs. In that strategy, common parts of a configuration can be proven only once and then feature-dependent parts are proven based on this first partial proof.

Variational Method Call

The second type of variation point on code level is the call of a method that can have different implementations due to varying feature compositions, i.e., a *variational method call*. We call a method variational if it is defined in more than one feature module. It is not necessary that an `original` call is used for a variational method call, because overriding a method without calling `original` also results in different implementations. We define a variational method as follows.

Definition 5.9: Variational Method varM (Bordis et al., 2022b)
We define the term *variational method* to describe a method *varM* that is defined in more than one feature module. In other words, there are at least two features $f_i \in \mathbb{F}$, $i \in \{1, \ldots, n\}$ such that $\mathtt{varM} \in \mathrm{impl}(f_i)$, and for these features f_i, there is at least one configuration $fc \in \mathrm{FM}$ *such that* $f_i \in fc$.

Example 5.3. *In the IntegerList product line, all shown methods are variational methods since they are all defined in more than one feature module. Therefore, their implementation and contract depends on the feature configuration. When implementing a new method using CbC, an abstract statement* `S` *can be refined to a method call to any of these variational methods (e.g., call to method* `sort` *in method* `push` *as shown in Section 5.1). To guarantee the correctness of this variational method call, all variants of the called method (callee) in composition with the calling method (caller) have to be considered.*

We define the refinement rule for a variational method call similar to the refinement rule for an `original` call with the two verification strategies in the side condition:

Definition 5.10: Variational Method Call Refinement Rule
Let the context of the abstract statement $\mathtt{S}$ be $\mathit{context}(\mathtt{S}) = (\mathrm{FM}, f, \mathtt{m})$. Then $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ *can be refined to* $\{\mathtt{P}\}\ \mathtt{b} := \mathtt{varM}(\mathtt{a}_1, \dots, \mathtt{a}_\mathtt{n})\ \{\mathtt{Q}\}$ iff one of the verifications strategies: *product-based composition* (Definition 5.16) or a *family-based encoding* of the whole method `varM` is performed and the method call refinement rule of Definition 2.2 is used and the respective side conditions hold.

5.2.3. Discussion

In the following, we discuss some characteristics of our variational specification concepts and observations that we made during our work on the feature-based CbC phase.

Uninterpreted Predicates and Refinement Rules in CbC. In this section, we defined the original predicates and the refinement rules for `original` calls and variational method calls with a side condition that references concrete verification strategies that we introduce in the next section. This means, that the interpretation of the side conditions on the specification and implementation of the method under construction in other feature modules and remains uninterpreted until all information that is needed for the chosen verification strategy has been defined. The original predicates, the `original` call refinement rule, and the variational method call refinement rule remain uninterpreted until they are resolved by either applying our family-based or product-based verification strategy to guarantee correctness. To resolve the original predicates and the new refinement rules in the product-based verification strategy, the method refinements in the features that are smaller according to the the feature composition than the feature of the current context need to be specified and for the family-based verification strategy even the whole product line needs to be specified. This limits the advantage of CbC that the refinement steps can be shown correct locally and directly when they are applied. While it is unavoidable to introduce the new refinement rules to be able to introduce variation points in the first place, having the original predicates might be discussable as they can appear in more specifications than just the `original` call or variational method call refinement rule. We argue that the original predicates are crucial, because otherwise the specification has to describe every possible behavior, which drastically increases the burden of the developer. The developer can decide if and how often the original predicates are used in the specification and therefore can control the degree of dependency with the disadvantage that precision of specifications may decrease because they need to be formulated more generally for all product variants. If no original predicate is used in the specifications, the side conditions of a CbC refinement step can be checked directly.

All in all, there is a trade-off between the flexibility of checking refinement steps locally and the modular reasoning introduced by the original predicates. The degree can be decided by the developer depending on the specific case. From experience, the original predicates are very useful and needed in software product line specifications to provide an expressive specification for all product variants in the product line.

Trade-Off between Precision and Increased Mental Effort Using Two Different Specification Techniques for Variability. While we argue that the original predicates are mandatory to reasonably specify a feature-oriented software product line, with variable predicates, we added another source of variability in the specifications. While simple variable predicates, such as `isSorted` from our running example, can easily be applied to larger case studies, the mental effort for defining and using variable predicates with many definition cases and complex presence conditions increases. Therefore, we recommend to define smaller predicates where precision for the specification is needed and use them in concert with other specification techniques. In the end, the developer can decide if and how often variable predicates are used and can balance the degree of having a more precise specifications and the difficulty to determine the cases of a variable predicate.

Abstraction of Variable Predicates. Using variable predicates to specify software product lines has the advantage that variable predicates abstract certain parts of a specification using descriptive names. This modularization increases readability and maintainability of specifications. The effect is similar to encapsulating certain code in functions, which is a standard practice in programming languages to achieve understandable and maintainable code. This abstraction has several advantages. First, it allows to define a variable predicate once and reuse this definition where needed without copying the definitions among specifications. Second, variable predicates can be used to specify the overall behavior without concretely expressing the logical meaning of a property. Later, the concrete definition of the variable predicate can be added. Third, when the implementation is changed or extended, it is not necessarily required to adapt the corresponding specification when a variable predicate was used. In certain cases, these changes can be covered by adapting the definitions of the variable predicate, for example by adding a new case.

Information Hiding. The aforementioned abstraction of variable predicates has the consequence that their definition is hidden and not directly viewable in the specification where it is used. Even though this comes with advantages, such as readability and maintainability, the developer also has to look into the definition of a variable predicate to understand the concrete definition, for example, when looking for a bug. The same holds for the original predicates where the information is hidden even more as it is defined indirectly over the order of the method implementations in the feature modules. This disadvantage of hiding information can be attenuated with tool support that facilitates finding and displaying the needed information, but a certain trade-off remains between abstraction for readability and hiding information. We argue that especially for long and complex specifications the advantages of modularizing and abstracting from the concrete definition benefit a specification.

5.3. Verification Approaches for Correct-by-Construction Software Product Lines

In Section 5.1, we identified four variation points: `original` calls and variational method calls in the code, the original predicates `\original_pre` and `\original_post`, and variable predicates in the variational specifications. These variation points cause the variability in the product line. In the previous section, we defined all constructs in a way that they cannot be interpreted yet. To make them interpretable, one of the verification strategies that we present in this section has to be performed. These make the uninterpreted variation points concrete and verifiable. In this section, we propose two different verification strategies for correct-by-construction software product lines that have been constructed with variational CbC from the previous section. In Section 5.3.1, we introduce the product-based verification strategy, which defines how original predicates, variable predicates, `original` and variational method call refinement rules can be composed and proven in a product-based way. In Section 5.3.2, we propose a family-based verification strategy relying on the generation of a metaproduct that simulates every possible product variant.

5.3.1. Product-based Verification

For our product-based verification strategy, we generate concrete side conditions for every relevant product variant of that method whenever one of the variation points (original predicates, variable predicates, `original` or variational call refinement rule) is reached to cover all relevant products from the product line. Since generating all valid product variants does not scale well due to the exponentially growing number of product variants with optional features, we define an optimized product-based verification strategy. To do so, we first need to define which feature sequences are *relevant*, i.e., the smallest set of feature sequences forming valid method variants for all variation points. Afterwards, we define contract composition to concretize variational specifications and we define refinement rules for `original` calls and variational method calls that concretize the side conditions in Definitions 5.8 and 5.10 for product-based verification.

Contract Composition

In Definition 5.2, we defined the original predicates `\original_pre` and `\original_post` with varying definitions that depend on the chosen verification strategy. For product-based verification strategies, contracts are composed. As already explained in Section 5.2.1, we defined original predicates on the basis of explicit contract refinement (Thüm et al., 2019). Our adaption of the definition of explicit contract refinement for CbC and original predicates is as follows:

Definition 5.11: Explicit Contract Refinement • (Bordis et al., 2022b)
Explicit contract refinement is a function $\bullet : \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C}$ defined over the set $\mathbb{C}$ of all possible contracts. Let the original contract of a method m be $c = \{\mathtt{P}\}\ \mathtt{m}\ \{\mathtt{Q}\}$, the refining contract $c' = \{\mathtt{P}'\}\ \mathtt{m}\ \{\mathtt{Q}'\}$, then we define the composed contract as $c'' = c' \bullet c = \{\mathtt{P}'\}\ \mathtt{m}\ \{\mathtt{Q}'\} \bullet \{\mathtt{P}\}\ \mathtt{m}\ \{\mathtt{Q}\} = \{\mathtt{P}''\}\ \mathtt{m}\ \{\mathtt{Q}''\}$ with $\mathtt{P}'' = \mathtt{P}'$ *where* `\original_pre` *is replaced by* P *and* `\original_post` *is replaced by* Q and $\mathtt{Q}'' = \mathtt{Q}'$ *where* `\original_pre` *is replaced by* P *and* `\original_post` *is replaced by* Q.

Contract composition is always performed for one method `m` independently. In feature-oriented programming, methods can be refined resulting in different implementations for the constructs in c and c'. Nevertheless, the signature of the refined method has to be maintained. There are two special constructs that are exclusively allowed to be used in postconditions, namely the reference to the value of a variable before the execution of method `m` (denoted as v^{old}) and the return value. Therefore, we forbid using predicate `\original_post` in the precondition of any starting triple. But, we want to allow the usage of `\original_post` in intermediate conditions, which are the postcondition for one Hoare triple and a precondition for another. For CbC, we use a method definition, where the return value has an explicit variable name (see Section 2.2). Consequently, the return value in the pre-/postconditions is used by explicitly using the name of the return variable defined in the signature such that there are no conflicts when composing two conditions that contain information about a return value. Other specification languages, like the JAVA MODELING LANGUAGE (JML) for JAVA (Leavens and Cheon, 2006), use a keyword for the return value. For that concept, our definition would have to be extended by replacing the keyword with the actual variable name that the return value is assigned to after calling a method. For the usage of variable values before method execution, we need to make an adaption to intermediate conditions. Generally, when the keyword `old` is used in the postcondition of an Hoare triple {P} S {Q}, it means that this is the value before executing the statement S. Therefore, it is used in an intermediate condition to refer to the value of a variable before executing the statement. When the next statement is executed, the reference to `old` has to be updated, because in the next postcondition the `old` keyword refers to the state before the second statement. Therefore, for every variable v that uses the keyword `old` in an intermediate condition, we update the value of v^{old} by adding $v = v^{old}$ to the precondition of the second abstract statement.

Example 5.4. *In Listing 5.1, we defined the intermediate condition* `M` *using* `\original_post`. *Using explicit contracting, the composed conditions for* `M` *are shown in Listing 5.5 using* $\texttt{data}^{\texttt{old}}$ *in their specifications. For* `M` *as postcondition of the* `original` *call refinement rule* ② *in Figure 5.2, old refers to the state of* `data` *before using the* `original` *call and is used to express that the* `original` *call only adds a new element to the array, but does not manipulate any other values. For* `M` *as precondition in the method call* ③*, the state of old has to be updated, because in the postcondition of this method call it references the state after the new element has been added by the* `original` *call. Therefore, it is implicitly added to the precondition of the method call* `sort` *that* `data` *is equal to* $\texttt{data}^{\texttt{old}}$. *The predicate* $\texttt{containsAll}(\texttt{data}, \texttt{data}^{\texttt{old}})$ *would then evaluate to true, because the equality condition says that* `data` *and* $\texttt{data}^{\texttt{old}}$ *have the same elements in the precondition.*

In product lines, it is common to refine a method more than once. This results in *sequences of contract compositions*. Such a sequence consists of features from a valid feature configuration and results in one composed contract for each method. For example, a method `a()` may have a different refinement sequence than method `b()` for the same feature configuration, as the methods might be implemented by different feature modules contained in the configuration. We formally define a contract composition sequence as follows.

Definition 5.12: Contract Composition Sequence $c_1 \bullet\ c_2 \bullet\ \ldots \bullet\ c_l$ **(Bordis et al., 2022b)**
We define a *contract composition sequence* as the contract c resulting from the contract composition of a sequence of n features $[f_1, ..., f_n]$. The composition is performed method-wise. Let c_k be the contract of method `m` which is contained in one or more feature modules $\texttt{m} \in impl(f_i)$ with $i \in \{1, ..., n\}$, $k \in \{1, ..., l\}$, *and* $l \leq n$, then c is composed as

$$c = c_1 \bullet\ c_2 \bullet\ \ldots \bullet\ c_l$$

resulting in a contract c of the form described in Definition 2.1. The contract composition mechanism • is explicit contract refinement as described in Definition 5.11.

It is important that method `m` does not necessarily have to be contained in every feature module of the sequence $[f_1, ..., f_n]$. Therefore, the number of composed contracts l can be less than or equal to the number of features n.

Example 5.5. *In Listing 5.4, we already gave the pre- and postcondition of method* `push` *in feature module* impl(*Limited*)*. In Listing 5.8, we now show the composed contract for feature configuration* $fc = [Base, Limited, Sorted]$ *with an intermediate result in the top box. The contract composition sequence for feature configuration* $fc = [Base, Limited, Sorted]$ *in the IntegerList product line of method* `push` *is defined*

```
P_Base•Limited  = data != null                                      Base • Limited
Q_Base•Limited  = (data^old.length < LIMIT) → (contains(data, newTop) &
                  containsAll(data^old, data))
```

```
P = data != null                                          Base • Limited • Sorted
P = ((data^old.length < LIMIT) → (contains(data, newTop) &
    containsAll(data, data^old))) & (isSorted(data))
```

Listing 5.8: Pre- and postcondition of method push in feature module impl(*Limited*) and composed contract for method push with $fc = [Base, Limited]$ and $fc = [Base, Limited, Sorted]$ (Bordis et al., 2022b). The feature-based specifications of method push in features *Base* and *Limited* is defined in Listing 5.4.

as: $c = c_{Base} \bullet c_{Limited} \bullet c_{Sorted}$. *The contract composition for postcondition* Q *of the contract of the* push *method starts with the composition of Base and Limited. Therefore, the predicate* \original_post *is replaced with the postcondition of feature Base. In the next step, the resulting postcondition* $\mathtt{Q}_{Base \bullet Limited}$ *is composed with the postcondition of feature Sorted:*

$$\mathtt{Q} = \mathtt{Q}_{Base \bullet Limited} \bullet \mathtt{Q}_{Sorted}$$

Original Predicates and Original Calls

Based on the context of a refinement rule, the relevant feature sequences for a variation point can be computed. This set only contains those feature sequences that form a valid product variant of the method that is currently refined for a specific variation point. The set of relevant feature sequences is given by the feature model, which explicitly defines dependencies between features and the order in which they are composed. The original predicates in the variational specifications and the original call in the code are resolved by composing specifications over the same set of relevant feature sequences. Consequently, we can define a set of *relevant feature sequences* for the original call refinement rule and the original predicates \original_pre and \original_post to guarantee correctness of the refinements. The following definition for the set of relevant feature sequences for original predicates and original calls is given by three insights:

1. Only feature configurations involving feature f of context(S) can contain relevant replacements for an original predicate or an original call.
2. Valid replacements can only be formed by features that are smaller than feature f which is a consequence of the feature composition order of feature model FM. The feature model determines the order in which the features are composed, or in other words, which features can actually be called by the original call or referred to by the original predicates.
3. Features that do not implement method m from context(S) do not alter the resulting composed contract.

Definition 5.13: Relevant Feature Sequences $c_f^{\mathtt{m}}$ for Original Predicates and the `Original` Call (Bordis et al., 2022b)
Let the context of the abstract statement `S` be context(`S`)=(FM, f, `m`). The set of features of a configuration fc that are smaller than feature f in the feature composition order of FM and implement method `m` is defined by $fc_{\mathtt{m}} = \{f' \in fc \mid \mathtt{m} \in \text{impl}(f') \textit{ and } f' < f\}$.
Then, the set of configurations of the feature model FM containing feature f projected onto the smaller features implementing method `m` is defined as $\text{FM}_f^{\mathtt{m}} = \{fc_{\mathtt{m}} \mid fc \in \text{FM}_f\}$.
We define the set of *relevant feature sequences for an original predicate or an `original` call* in context(`S`) as

$$c_f^{\mathtt{m}} = \{[fc_f^{\mathtt{m}}] \mid fc_f^{\mathtt{m}} \in \text{FM}_f^{\mathtt{m}}\}$$

Based on the relevant feature sequences, we propose a refinement rule for the `original` call. It is based on the method call refinement rule introduced in the background chapter in Section 2.2. To guarantee the correctness for an `original` call, the composed contracts of all relevant feature sequences have to comply with the contract of the calling statement. For the product-based verification strategy, we generate the side condition for this refinement rule using contract composition as described in Section 5.2.1 over the set of relevant feature sequences for original predicates and the `original` call.

In every CbC refinement rule, including the basic set of Definition 2.2, a Hoare triple of the form $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ is refined. As described in Section 5.2.1 the conditions `P` and `Q` can contain the predicates `\original_pre` and `\original_post`. We treat `\original_pre` and `\original_post` like uninterpreted predicates and a side condition using `P` or `Q` can only be checked, if there are no more uninterpreted predicates in the specification. To resolve the uninterpreted predicates, contract composition as described in Section 5.2.1 can be applied over the set of relevant feature sequences from Definition 5.13. The result of this step are multiple side conditions for the same refinement step that all have to be checked to guarantee correctness for all relevant product variants. For the `original` call refinement rule, we use the notation $\hat{\mathtt{P}}_i/\hat{\mathtt{Q}}_i$ to denote the resolved versions of the conditions $\hat{\mathtt{P}}$ and $\hat{\mathtt{Q}}$ of the refined triple $\{\hat{\mathtt{P}}\}\ \mathtt{b} := \mathtt{original}(\mathtt{a}_1, ..., \mathtt{a}_n)\ \{\hat{\mathtt{Q}}\}$ over the set of relevant feature sequences. As the original predicates can also be used in the basic set of CbC refinement rules, the product-based generation of side conditions using contract composition has to be applied in every refinement step that uses `\original_pre` and `\original_post` in the specification. Consequently, the basic set of refinement rules has to be adapted analogously to the definition of the `original` call refinement rule using $\hat{\mathtt{P}}_i/\hat{\mathtt{Q}}_i$ in the side condition.

We define the `original` call refinement rule based on Definition 5.13 for the set of relevant feature sequences as follows:

```
P₁ = data != null                                                    Base
Q₁ = contains(data, newTop) & containsAll(data^old, data)

P₂ = data != null                                         Base • Limited
Q₂ = (data^old.length < LIMIT) → (contains(data, newTop) &
    containsAll(data^old, data))
```

Listing 5.9: Composed contract for the `original` call refinement rule in method `push` in feature *Sorted* (Bordis et al., 2022b).

```
P̂₁ implies P₁[newTop\newTop] and Q₁[newTop\newTop] implies M̂₁
= (data != null) implies (data != null) and (contains(data, newTop) &
  containsAll(data^old, data)) implies (contains(data, newTop) &
  containsAll(data^old, data))

P̂₂ implies P₂[newTop\newTop] and Q₂[newTop\newTop] implies M̂₁
= (data != null) implies (data != null) and ((data^old.length < LIMIT) → (
  contains(data, newTop) & containsAll(data^old, data))) implies ((data^old
  .length < LIMIT) → (contains(data, newTop) & containsAll(data^old,
  data)))
```

Listing 5.10: Side conditions for `original` call refinement rule in method `push` in feature *Sorted* (Bordis et al., 2022b).

Definition 5.14: Original Call Refinement Rule (Product-based) (Bordis et al., 2022b)
Let the context of the abstract statement $\mathtt{S}$ be $\mathrm{context}(\mathtt{S}){=}(\mathrm{FM}, f, \mathtt{m})$. Then $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ can be refined to $\{\hat{\mathtt{P}}\}\ \mathtt{b} := \mathtt{original}(\mathtt{a}_1, ..., \mathtt{a}_n)\ \{\hat{\mathtt{Q}}\}$ iff for all contracts c_i

$$c_i{=}\{\mathtt{P}_i\}\ \mathtt{return\ r\ m(param\ p_1, .., p_n)}\ \{\mathtt{Q}_i\}$$

composed over the set of relevant feature sequences $c_f^{\mathtt{m}}$ as defined in Definition 5.13, it holds that

$$\hat{\mathtt{P}}_i\ \mathit{implies}\ \mathtt{P}_i[\mathtt{p_j}\backslash\mathtt{a_j}]\ \mathit{and}\ \mathtt{Q}_i[\mathtt{p_j^{old}}\backslash\mathtt{a_j^{old}}, \mathtt{r}\backslash\mathtt{b}]\ \mathit{implies}\ \hat{\mathtt{Q}}_i$$

Example 5.6. *In Figure 5.2, we already showed the implementation of method* `push` *in feature module* impl(*Sorted*) *with an* `original` *call refinement rule in step* ②. *The context is defined as* context(`S1`) = (FM, *Sorted*, `push`) *and the resulting set of relevant feature sequences is* $c_{Sorted}^{\mathtt{push}} = \{[Base], [Base, Limited]\}$. *These feature sequences need to be considered when applying the* `original` *call refinement rule. First, the pre- and postcondition (*$\hat{\mathtt{P}}$ *and* $\hat{\mathtt{Q}}$ *in the definition of the refinement rule and* `P` *and* `M` *in the example) need to be composed, as they contain the predicates* `\original_pre` *and* `\original_post` *(see Listing 5.5). Second, the pre- and postconditions* $\mathtt{P}_i$ *and* $\mathtt{Q}_i$ *have to be retrieved using contract composition over the relevant feature sequences. As there are two elements in the set of relevant feature sequences, we can generate two pre- and postcondition pairs which we display in*

Listing 5.9. The conditions `P`$_1$ *and* `Q`$_1$ *are the concrete contract for the first method variant and* `P`$_2$ *and* `Q`$_2$ *the contract for the second method variant of Listing 5.5. We show the concrete side conditions for the product-based application of the* `original` *call refinement rule* ② *in Listing 5.10. In this case, the refinement is correct, as the composed pre- and intermediate condition are exactly the same. This is because the specifications* `P` *and* `M` *only consist of* `\original_pre` or `\original_post`, *respectively.*

Variational Method Call

In Section 5.2.2, we defined variational method calls as method calls that have varying implementations in different product variants of the software product line. Consequently, similar to an `original` call, the implementation and contract of a variational method also depend on a specific feature configuration. Therefore, to guarantee correctness for the whole product line, we define the *set of relevant feature sequences for variational method calls* as well. This set is defined by two insights:

1. Only feature configurations that involve feature f from context(`S`) (abstract statement `S` in the calling method is refined to the variational method call) can contain relevant replacements for a variational method call.
2. Features that do not implement method `varM` do not alter the resulting composed contract.

Definition 5.15: Relevant Feature Sequences for Variational Method Call $c_f^{\texttt{varM}}$ (Bordis et al., 2022b)
Let the context of an abstract statement `S` be context(`S`)=(FM, f, `m`), and let method `varM` be a variational method that is called by method `m` as a refinement step from the abstract statement `S`. Method `varM` is defined in the feature modules impl(f') and these features f' are contained in a feature configuration fc of the valid feature configurations for feature f FM$_f$ ($\texttt{varM} \in \{impl(f') \mid f' \in fc\}\ for\ fc \in \text{FM}_f$).
The set of features of a given configuration fc implementing `varM` is defined by $fc_{\texttt{varM}} = \{f'' \in fc \mid \texttt{varM} \in \text{impl}(f'')\}$.
Then, the set $\text{FM}_f^{\texttt{varM}}$ is the set of configurations of a feature model FM containing feature f projected onto the features implementing `varM`. We formally define it as $\text{FM}_f^{\texttt{varM}} = \{fc_{\texttt{varM}} \mid fc \in \text{FM}_f\}$. We define the set of *relevant feature sequences for a variational method call* in the context(`S`) as

$$c_f^{\texttt{varM}} = \{[fc_f^{\texttt{varM}}] \mid fc_f^{\texttt{varM}} \in \text{FM}_f^{\texttt{varM}}\}$$

The first insight above is the same as for original predicates and `original` calls. The second insight above is very similar to the third insight for original predicates and `original` calls with the difference that in this case only features that implement the called method `varM` are important. Consequently, feature f from context(`S`) is not necessarily contained in the resulting feature sequences from the set of relevant feature sequences, if feature f does not implement method `varM`. Another difference is that all features, no matter if they are smaller or greater than feature f, can be part of the valid replacements. This is due to the fact that the refinement sequence of the called method `varM` is independent of the refinement sequence of method `m` from context(`S`). That method `varM` exists and can be called has to be guaranteed by the feature model and is orthogonal to the problem considered here.

The variational method call refinement rule can be defined analogously to the `original` call refinement rule, but uses a different set of relevant feature sequences in addition to the one for original predicates and `original` calls. Furthermore, the side condition contains a check whether the combination of the relevant feature sequences for original predicates and `original` calls that is used to compose the specifications of the refined triple and the ones for variational method calls is a partial configuration of the feature model (see Definition 2.4), because we only want to guarantee correctness for valid products. We define the variational method call refinement rule using the set of relevant feature sequences from Definition 5.15.

Definition 5.16: Variational Method Call Refinement Rule (Product-based) (Bordis et al., 2022b)
Let the context of the abstract statement $\mathtt{S}$ be context($\mathtt{S}$) $=$ (FM, f, $\mathtt{m}$). Then $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ *can be refined to* $\{\hat{\mathtt{P}}\}\ \mathtt{b} := \mathtt{varM}(\mathtt{a}_1, ..., \mathtt{a}_n)\ \{\hat{\mathtt{Q}}\}$ *iff* for all $k_l \in c_f^{\mathtt{m}}$ and the composed specifications $\hat{\mathtt{P}}_l$ and $\hat{\mathtt{Q}}_l$ it holds that for all contracts c_i

$$c_i = \{\mathtt{P}_i\}\ \mathtt{return\ r\ varM(param\ p}_1, .., \mathtt{p}_n)\ \{\mathtt{Q}_i\}$$

composed over the relevant feature sequences for variational method calls $c_f^{\mathtt{varM}}$ from Definition 5.15, it holds that
isPartialConfig$(\mathrm{FM}, k_l \cup f \cup c_i)$ *implies* $(\hat{\mathtt{P}}_l$ *implies* $\mathtt{P}_i[\mathtt{p}_\mathtt{j} \backslash \mathtt{a}_\mathtt{j}]$ *and* $\mathtt{Q}_i[\mathtt{p}_\mathtt{j}^{\mathtt{old}} \backslash \mathtt{a}_\mathtt{j}^{\mathtt{old}}, \mathtt{r} \backslash \mathtt{b}]$ *implies* $\hat{\mathtt{Q}}_l)$

Example 5.7. *To showcase a variational method call, we use our previous example of the IntegerList product line of Section 5.1 again. We show the implementation of method* `push` *in feature* Sorted *in Figure 5.2 with a method call of method* `sort` *in step* ③*. If method* `sort` *was defined in only one feature module, the method call refinement rule from the basic set of CbC refinement rules would be applied. As method* `sort` *is defined by two features, namely* Increasing *and* Decreasing, `sort` *is a variational method and therefore, we apply the variational method call refinement rule. The context is defined as* context(`S2`) $=$ (FM, *Sorted*, `push`) *and the resulting set of relevant feature sequences for the variational method call* $c_{Sorted}^{\mathtt{sort}} = \{[Increasing], [Decreasing]\}$*. As the pre- and postcondition of the refined Hoare triple (*$\hat{\mathtt{P}}$ *and* $\hat{\mathtt{Q}}$ *in the refinement rule definition and* `M` *and* `Q` *in the example) contain predicate* `\original_post`*, we first need to compose the pre-/postconditions over the set of relevant feature sequences for original predicates (*$c_{Sorted}^{\mathtt{push}} = \{[Base], [Base, Limited]\}$*). The results are shown in Listing 5.11. For each of the generated pre- and postcondition pairs, we now have*

```
{M} data := sort(data) {Q}
M̂1 = contains(data, newTop) & containsAll(data^old, data)                          Base
Q̂1 = contains(data, newTop) & containsAll(data^old, data) & isSorted(data)

M̂2 = (data^old.length < LIMIT) → (contains(data, newTop) &
     containsAll(data^old, data))                                        Base • Limited
Q̂2 = (data^old.length < LIMIT) → (contains(data, newTop) &
     containsAll(data^old, data)) & isSorted(data)

composed contracts of int[] r sort(int[] a):
P1 = true                                                                  Increasing
Q1 = containsAll(a^old, r) & (\forall int i; i>=0 & i<= r.length-1; r[i] <= r[i+1])

P2 = true                                                                  Decreasing
Q2 = containsAll(a^old, r) & (\forall int i; i>=0 & i<= r.length-1; r[i] >= r[i+1])
```

Listing 5.11: Composed contract for variational method call refinement rule in method `push` in feature *Sorted* (Bordis et al., 2022b). The variational method that is called is method `sort`.

to check in a second step, whether every variant of method `sort` *complies with this derived specification. To omit the generation of invalid feature configurations, the predicate* isPartialConfig *is used in the side condition of the variational method call refinement rule. The concrete side conditions for the variational method call refinement rule are displayed in Listing 5.12. In the next subsection, we complete the example by resolving the variable predicate* `isSorted`.

Variable Predicates

In Definition 5.3, we define variable predicates as first-order logic predicates that have mutiple definitions each tied to a presence condition over the features of the feature model. In the product-based verification strategy that we propose in this section, we only verify the relevant feature sequences of each variation point. The relevant feature sequences for variable predicates are determined by the definition space of a variable predicate as defined in Definition 5.5.

Example 5.8. *In our running example of the IntegerList product line, we use the variable predicate* `isSorted` *that is defined with an increasing and a decreasing order for array* `data` *(see Listing 5.3). The increasing order is defined for presence condition Sorted $\wedge$ Increasing and the decreasing order is defined for the presence condition Sorted $\wedge$ Decreasing. The set of relevant feature sequences for a variable predicate is defined as the definition space of a variable predicate (Definition 5.5), which in this case is the set* $c^{\texttt{isSorted}} = \{[Base, Limited, Sorted, Increasing], [Base, Sorted, Increasing], [Base, Limited, Sorted, Decreasing], [Base, Sorted, Decreasing]\}$. *The variable*

```
isPartialConfig(FM, [Base,Sorted,Increasing]) & ($\hat{M}_1$ implies $P_1$[a\data] and $Q_1$[$a^{old}$\$data^{old}$,r\data] implies $\hat{Q}_1$)
= true & (contains(data, newTop) & containsAll($data^{old}$, data)) implies (true) and (containsAll($data^{old}$, data) & (\forall int i; i>=0 & i<= data.length-1; data[i] <= data[i+1])) implies (contains(data, newTop) & containsAll($data^{old}$, data) & isSorted(data))

isPartialConfig(FM, [Base,Sorted,Decreasing]) & ($\hat{M}_1$ implies $P_2$[a\data] and $Q_2$[$a^{old}$\$data^{old}$,r\data] implies $\hat{Q}_1$)
= true & (contains(data, newTop) & containsAll($data^{old}$, data)) implies (true) and (containsAll($data^{old}$, data) & (\forall int i; i>=0 & i<= data.length-1; data[i] >= data[i+1])) implies (contains(data, newTop) & containsAll($data^{old}$, data) & isSorted(data))

isPartialConfig(FM, [Base,Limited,Sorted,Increasing]) & ($\hat{M}_2$ implies $P_1$[a\data] and $Q_1$[$a^{old}$\$data^{old}$,r\data] implies $\hat{Q}_2$)
= true & ($data^{old}$.length < LIMIT) → (contains(data, newTop) & containsAll($data^{old}$, data)) implies (true) and (containsAll($data^{old}$, data) & (\forall int i; i>=0 & i<= data.length-1; data[i] <= data[i+1])) implies (($data^{old}$.length < LIMIT) → (contains(data, newTop) & containsAll($data^{old}$, data)) & isSorted(data))

isPartialConfig(FM, [Base,Limited,Sorted,Decreasing]) & ($\hat{M}_2$ implies $P_2$[a\data] and $Q_2$[$a^{old}$\$data^{old}$,r\data] implies $\hat{Q}_2$)
= true & ($data^{old}$.length < LIMIT) → (contains(data, newTop) & containsAll($data^{old}$, data)) implies (true) and (containsAll($data^{old}$, data) & (\forall int i; i>=0 & i<= data.length-1; data[i] >= data[i+1])) implies (($data^{old}$.length < LIMIT) → (contains(data, newTop) & containsAll($data^{old}$, data)) & isSorted(data))
```

Listing 5.12: Product-based evaluation of the side conditions for variational method call refinement rule in method `push` in feature *Sorted* (Bordis et al., 2022b). Variable predicate `isSorted` in color teal is resolved in Listing 5.13.

predicate `isSorted` *is used in the variational method call refinement step from Example 5.7. In this example, the relevant feature sequences of the variational method call* `sort` *are the same as for the variable predicate* `isSorted`*. Therefore, we can directly evaluate the variable predicate for the individual feature configurations, and get the complete side conditions in Listing 5.13. If this was not the case, the feature sequences of both sets of relevant feature sequences, of the variable predicate and the applied refinement rule, need to be composed and checked.*

Proof of Soundness

In this subsection, we reason about the soundness of our product-based verification strategy which means that all programs in a product line are correct if the product line is constructed using our refinement rules. Concretely, we consider this on a per-method basis meaning that the product line is correct if all methods that can

```
1. [Base, Sorted, Increasing]: true & (contains(data, newTop) &
   containsAll(data^old, data)) implies (true) and (containsAll(data^old,
   data) & (\forall int i; i>=0 & i<= data.length-1; data[i] <= data[i
   +1])) implies (contains(data, newTop) & containsAll(data^old, data) &
   (\forall int i; 0 <= i & i < data.length-1; data[i] <=
   data[i+1])
2. [Base, Sorted, Decreasing]: true & (contains(data, newTop) &
   containsAll(data^old, data)) implies (true) and (containsAll(data^old,
   data) & (\forall int i; i>=0 & i<= data.length-1; data[i] >= data[i
   +1])) implies (contains(data, newTop) & containsAll(data^old, data) &
   (\forall int i; 0 <= i & i < data.length-1; data[i] >=
   data[i+1])
3. [Base, Limited, Sorted, Increasing]: true & (data^old.length < LIMIT)
    → (contains(data, newTop) & containsAll(data^old, data)) implies (
   true) and (containsAll(data^old, data) & (\forall int i; i>=0 & i<=
   data.length-1; data[i] <= data[i+1])) implies ((data^old.length < LIMIT
   ) → (contains(data, newTop) & containsAll(data^old, data)) & (\forall
   int i; 0 <= i & i < data.length-1; data[i] <= data[i+1])
4. [Base, Limited, Sorted, Decreasing]: true & (data^old.length < LIMIT)
    → (contains(data, newTop) & containsAll(data^old, data)) implies (
   true) and (containsAll(data^old, data) & (\forall int i; i>=0 & i<=
   data.length-1; data[i] >= data[i+1])) implies ((data^old.length < LIMIT
   ) → (contains(data, newTop) & containsAll(data^old, data)) & (\forall
   int i; 0 <= i & i < data.length-1; data[i] >= data[i+1])
```

Listing 5.13: Product-based evaluation of variational method call refinement step with variable predicate `isSorted` being resolved.

be generated for all valid feature configurations satisfy their pre- and postconditions. We assume that the refinement rules from Definition 2.2 are sound (Morgan, 1990). Hence, for soundness we only need to consider the refinement rules introduced in Definitions 5.14 and 5.16.

We establish a lemma for the relevant feature sequences used in both refinement rules and the relevant feature sequences for variable predicates. We prove that the set of relevant feature sequences for original predicates and the `original` call (see Definition 5.13), the variational method call (see Definition 5.15), and variable predicates (see Definition 5.5) contain all valid replacements and only those with a proof by contradiction.

Lemma 5.1 (Relevant Feature Sequences)**.** *If a triple* $\{\mathtt{P}\}\mathtt{S}\{\mathtt{Q}\}$ *is refined to either* $\{\mathtt{P}\}\mathtt{b} := \mathtt{original}(\mathtt{a}_1, \dots, \mathtt{a}_n)\{\mathtt{Q}\}$ *or* $\{\mathtt{P}\}\mathtt{b} := \mathtt{varM}(\mathtt{a}_1, \dots, \mathtt{a}_n)\{\mathtt{Q}\}$ *using the* `original` *call refinement rule (Definition 5.14) or the variational method call refinement rule (Definition 5.16), there is no other valid replacement for* `original` *predicates and variable predicates in* `P` *and* `Q`, `original(...)` *or* `varM(...)` *than defined in the set of relevant feature sequences from Definition 5.13, Definition 5.15, or Definition 5.5, respectively.*

Proof of the `original` *Call and Original Predicates.* Assume, the set $c^f_{\mathtt{m}}$ of relevant feature sequences as defined in Definition 5.13 is not complete.

Let the context of the `original` call refinement rule be context(`S`)=(FM, *f*, `m`). Then there is a feature configuration fc' that forms a valid replacement for the `original` call `m′` with specifications `P′` and `Q′` that form valid replacements for the original predicates in `P` and `Q`, but is not contained in $c^f_{\mathtt{m}}$. The configuration fc' consists of a sequence of features $fc' = [f_1, ..., f_n]$ *with* $f_i < f, i \in 1, ..., n$. Because of the definition of the `original` call (Apel et al., 2013b) and because the original predicates analogously to the `original` call for specifications, all features f_i have to be smaller than feature *f*. The only configurations that fulfill this condition and are not contained in $c^f_{\mathtt{m}}$, are configurations that contain at least one feature that does not implement method `m`. However, the contract of the replacing method `m′` is the result of composing the contracts of the features implementing method `m` in fc'. Consequently, features that do not implement method `m` do not alter the resulting contract. This causes a contradiction and discounts the hypothesis. □

Proof of the Variational Method Call. Assume, the set c of relevant feature sequences as defined in Definition 5.15 is not complete.

Let the context of the variational method call refinement rule be context(`S`)=(FM, *f*, `m`). Then there is a feature configuration fc' that forms a valid replacement for the variational method `varM`, but is not contained in c. Because of the context of the refinement rule, configurations that do not contain feature f can not form a valid replacement for `varM`. The only configurations that fulfil this condition and are not contained in c are configurations that contain at least one feature that does not implement method `varM`. However, the contract of `varM` is the result of composing the contracts of the features implementing `varM` in fc'. Consequently, features that do not implement method `varM`, do not alter the resulting contract and can therefore be considered as equal to a configuration that does not contain the features not implementing `varM`. This causes a contradiction and discounts the hypothesis. □

Proof for Variable Predicates. Assume, the set $c^{\mathtt{varP}}$ of relevant feature sequences as defined in Definition 5.5 is not complete.

Let the context of the refinement rule where `varP` is used in the specification be context(`S`)=(FM, *f*, `m`). Then there is a feature configuration fc' that forms a valid replacement for variable predicate `varP`, but is not contained in $c^{\mathtt{varP}}$. There are two cases where a feature configuration is not contained in $c^{\mathtt{varP}}$. 1) Feature configurations that do not fulfill any of the presence conditions $PC_1, ..., PC_m$ of `varP` (Definition 5.5). In that case, feature configuration fc' cannot form a valid replacement for `varP`. 2) The set of feature configurations that need to be checked for context(`S`) (FM$_f$) where `varP` is used, exceeds the definition space of `varP` and therefore, `varP` cannot be resolved. This case is forbidden by Definition 5.6. This causes a contradiction and discounts the hypothesis.

Theorem 5.2 is the soundness of the CbC-based refinement technique and it is equally expressive as classical post-hoc Hoare logic verification.

Theorem 5.2 (Soundness of the Product-Based Verification Strategy)**.** *Let* FM *be a feature model, f a feature, and* `m` *a method implemented in* impl(f)*. If the starting triple (the contract) of method* `m` $\{\mathtt{P}\}\mathtt{S}\{\mathtt{Q}\}$ *with abstract statement* `S` *and* context(`S`) = (FM, f, m) *is refined to* $\{\hat{\mathtt{P}}\}\mathtt{C}\{\hat{\mathtt{Q}}\}$ *with resolvable* `original` *predicates in* $\hat{\mathtt{P}}$ *and* $\hat{\mathtt{Q}}$ *and a concrete implementation* `C` *following the refinement rules in Definition 2.2 and the rules that have been proposed in Section 5.3.1, then for all valid feature configurations fc of feature model* FM *which contain method* **m**, $\{\hat{\mathtt{P}}\}\mathtt{C}\{\hat{\mathtt{Q}}\}$ *holds.*

Proof. We show the above result by structural induction over the refinements. We assume that the rules presented in Section 2.2 already satisfy the result when they do not use the original predicates in specifications. Hence, it suffices to consider the `original` call and variational method call rules introduced in Section 5.3 and original predicates and variable predicates as two cases for all refinement rules.

- *Original Predicates:* Using `\original_pre` or `\original_post` introduces uninterpreted predicates in the specifications $\hat{\mathtt{P}}$ and $\hat{\mathtt{Q}}$ of a Hoare triple $\{\hat{\mathtt{P}}\}\ \mathtt{S}\ \{\hat{\mathtt{Q}}\}$ that are resolved by contract composition over the set of relevant feature sequences. By Lemma 5.1, the set of relevant feature sequences contains all possible replacements for the original predicates and only those. Hence, we can compose the specifications for each element of the relevant feature sequences and apply any refinement rule. The soundness of the basic refinement rules already holds, which closes this case of the proof.

- *Original Call Rule:* By applying the `original` call refinement rule, an abstract statement `S` is refined to $\{\hat{\mathtt{P}}\}\mathtt{b} := \mathtt{original}(\mathtt{a}_1, ..., \mathtt{a}_n)\{\hat{\mathtt{Q}}\}$. The `original` call is a placeholder for a method call that is resolved via the set of relevant feature sequences determined by context(`S`). By Lemma 5.1, the set of relevant feature sequences contains all possible replacements for the `original` call and only those. Hence, we can apply the method call refinement rule for each element of the relevant feature sequences. For the method call rule, the soundness result already holds and the treatment of uninterpreted predicates in $\hat{\mathtt{P}}$ and $\hat{\mathtt{Q}}$ has been shown in the first case of this proof, which closes this case of the proof.

- *Variational Method Call Rule:* By applying the variational method call refinement rule, an abstract statement `S` is refined to $\{\hat{\mathtt{P}}\}\mathtt{b} := \mathtt{varM}(\mathtt{a}_1, ..., \mathtt{a}_n)\{\hat{\mathtt{Q}}\}$. Thereby, `varM(...)` is a method call that can have different implementations which are resolved by the set of relevant feature sequences as defined in Definition 5.15. By Lemma 5.1, the set of relevant feature sequences contains all possible replacements for the variational method call and only those. The specifications $\hat{\mathtt{P}}$ and $\hat{\mathtt{Q}}$ are resolved using contract composition over the set of relevant feature sequences for original predicates which has been considered in the first case of this proof. To ensure the validity of both sets of relevant feature sequences according to the feature model FM, the predicate `isPartialConfig(FM, fc)` from Definition 2.4 is used. Hence, we can apply the method call refinement rule for each element of the relevant feature sequences if it forms at least a partial configuration in the product line with the feature sequence resolving the specifications $\hat{\texttt{P}}$ and $\hat{\texttt{Q}}$. For the method call rule, the soundness result already holds, which closes this case of the proof.

- *Variable Predicate:* Using a variable predicate `varP` introduces a predicate with m definitions that each depend on a presence condition $PC_1, ..., PC_m$ in the specifications $\hat{\texttt{P}}$ and $\hat{\texttt{Q}}$ of a Hoare triple $\{\hat{\texttt{P}}\}$ S $\{\hat{\texttt{Q}}\}$ that are resolved by valuating the presence conditions over the set of relevant feature sequences $c^{\texttt{varP}}$. By Lemma 5.1, the set of relevant feature sequences contains all possible replacements for a variable predicate and only those. Hence, we can evaluate the specifications for each element of the relevant feature sequences and apply any refinement rule. The soundness of the basic refinement rules already holds, which closes this case of the proof. □

5.3.2. Family-Based Verification

As an alternative to the product-based verification strategy presented in the previous subsection, we propose a family-based verification strategy. In contrast to product-based verification strategies, family-based verification strategies aim to express the behavior of all products in a single metaproduct. For that purpose, feature flags are oftentimes used to distinguish the different behaviors associated to features. The transformation from variational CbC with uninterpreted variation points (Section 5.2) to a family-based metaproduct is performed refinement step-wise for all cbcmethods in the feature modules of the product line. Consequently, the metaproduct consists of a family-encoded CbC method for each method introduced in the product line where every variation point has been encoded in a family-based way. As the encoded refinement steps in the metamethods do not contain any uninterpreted variations points, the correctness can be guaranteed by checking the side conditions of the applied refinement rules from Definition 2.2. To transform the variational CbC product line implementation into the metaproduct, implicit aspects of the variability, like which code is executed if a certain feature is selected, have to be made explicit. In general, three aspects of variability have to be encoded: the specification, the implementation in the feature modules, and the feature model. With this family-based verification strategy, it is possible to only conduct one proof per refinement step, while with the product-based verification strategy multiple proofs (for each relevant feature configuration) have to be conducted per refinement step.

Encoding the Specification

Since we want to guarantee that the metaproduct fulfills its specification, we need to transform the variational specifications into one large metaspecification. Thüm et al. (2012) proposed an approach to encode the variability of specifications for post-hoc verification with JML specifications. We adapt their encoding for our refinement-based CbC approach. To indicate the selection of a feature, a new variable is introduced for each feature. These feature flags are used to differentiate between different behaviors in the specification depending on the selection or deselection of a feature. The metaspecification is created by transforming every original and variable predicate method-wise for every method introduction in one of the feature modules. The goal is to create a specification for the different behaviors of a method according to different feature selections without any uninterpreted original or variable predicates.

Original Predicates. The metaspecification of a method `m` is encoded recursively for all features implementing method `m` from biggest (f_n) to smallest feature (f_0) according to the feature composition order. Starting with the biggest feature f_n, the pre- and postcondition are set to $\mathtt{P} = (f_n \;\rightarrow\; \mathtt{P}_{f_n})\;\&\;(\neg f_n \;\rightarrow\; \texttt{\textbackslash original\_pre})$ and $\mathtt{Q} = (f_n \;\rightarrow\; \mathtt{Q}_{f_n})\;\&\;(\neg f_n \;\rightarrow\; \texttt{\textbackslash original\_post})$. Thereby, $\mathtt{P}_{f_n}$ and $\mathtt{Q}_{f_n}$ are the feature-based pre- and postcondition of method `m` in the feature module of feature f_n and therefore also can contain additional original predicates. Next, all original predicates in `P` and `Q` are replaced by $(f_{n-1} \;\rightarrow\; \mathtt{P}_{f_{n-1}})\;\&\;(\neg f_{n-1} \rightarrow \texttt{\textbackslash original\_pre})$ for `\original_pre` and $(f_{n-1} \;\rightarrow\; \mathtt{Q}_{f_{n-1}})\;\&\;(\neg f_{n-1} \;\rightarrow\; \texttt{\textbackslash original\_post})$ for `\original_post` with the next smaller feature f_{n-1}. This procedure is repeated until the smallest feature f_0 is reached. For the smallest feature, the implication for when that feature is not selected is omitted and the original predicates are replaced by $(f_0 \;\rightarrow\; \mathtt{P}_{f_0})/(f_0 \;\rightarrow\; \mathtt{Q}_{f_0})$. As $\mathtt{P}_{f_0}$ and $\mathtt{Q}_{f_0}$ are not allowed to contain further original predicates, the encoding of the specifications is finished. We formally define a replacement operator for original predicates in a Hoare triple $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ as follows:

Definition 5.17: Original Predicate Transformation (Bordis et al., 2022b)
Let the context be context(`S`)=(FM, *f*, `m`) and *f'* the next smaller feature with `m` $\in$ impl(f').
Then `\original_pre` and `\original_post` in `P` and `Q` can be replaced by:

1. $$\mathit{replace}(\texttt{\textbackslash original\_pre}) = \begin{cases} (f \;\rightarrow\; \mathtt{P}_f), & \textit{if } f \textit{ is the smallest feature impl. } \mathtt{m} \\ (f \;\rightarrow\; \mathtt{P}_f) \wedge (\neg f \;\rightarrow\; \texttt{\textbackslash original\_pre}), & \textit{otherwise} \end{cases}$$

2. $$\mathit{replace}(\texttt{\textbackslash original\_post}) = \begin{cases} (f \;\rightarrow\; \mathtt{Q}_f), & \textit{if } f \textit{ is the smallest feature impl. } \mathtt{m} \\ (f \;\rightarrow\; \mathtt{Q}_f) \wedge (\neg f \;\rightarrow\; \texttt{\textbackslash original\_post}), & \textit{otherwise} \end{cases}$$

Repeat replacement for next smaller feature *f'*.

Variable Predicates. To encode a variable predicate in a family-based way, an implication of the presence conditions using feature flags to the different definitions can be formed: $PC_1 \rightarrow formula_1 \ \wedge \ ... \ \wedge \ PC_m \rightarrow formula_m$. We formally define a replacement operator for variable predicates in a Hoare triple {`P`} `S` {`Q`} as follows:

Definition 5.18: Variable Predicate Transformation
Let `varP` be a variable predicate with m definitions $formula_1, ..., formula_m$ that are connected to corresponding presence conditions $PC_1, ..., PC_m$, and let the context where `varP` is used be context(`S`)=(FM, *f*, `m`).
Then `varP` in `P` and `Q` can be replaced by:

$$replace(\texttt{varP}) = PC_1 \rightarrow formula_1 \ \wedge \ ... \ \wedge \ PC_m \rightarrow formula_m$$

Example 5.9. *To generate a metaproduct for the IntegerList product line, its specification has to be encoded variation point-wise for each method forming metaspecifications for each metaproduct (i.e., metaspecifications for metaproduct* `push`*, metaspecifications for metaproduct* `sort`*, ...). We present the encoded metaspecification for the contract (precondition* `P` *and postcondition* `Q`*) of method* `push` *in Listing 5.6. The features that implement* `push` *are* Base, Limited, *and* Sorted. *As* Sorted *is the greatest feature in the feature composition order of the feature model, the pre- and postconditions start with implications for the cases that* Sorted *is selected and that Sorted is not selected. The original predicates are then resolved recursively according to Definition 5.17. As* Base *is the smallest feature implementing method* `push`*, only the case for when feature Base is selected is inserted. Whenever a variable predicate is used, it is replaced by inserting all cases with implications of presence condition PC_i to the corresponding definition $formula_i$. In the example, variable predicate* `isSorted` *is replaced by its definitions for the increasing and decreasing order with presence conditions $Sorted \wedge Increasing$ and $Sorted \wedge Decreasing$.*

Encoding of Feature Module Implementations

The transformation of feature module implementations into a metaproduct was first proposed by Apel et al. (2011) and used by Thüm et al. (2012) to encode feature-oriented product lines implemented with Java for post-hoc verification. We adapt the approach of Thüm et al. (2012) for the encoding of feature module implementations for CbC software product lines. The encoding is performed for each `original` call refinement rule in each method in isolation. The differentiation between different behaviors due to a feature selection is achieved by applying the selection refinement rule (an equivalent in Java are if-else blocks) with a feature flag in the guard before inserting the CbC refinement steps implementing a method in a feature module.

The metaimplementation of a method `m` is encoded iteratively for all features implementing method `m` from biggest (f_n) to smallest feature (f_0) according to the feature composition order. The starting point is a Hoare triple {`P`} `S` {`Q`} with pre- and postcondition `P` and `Q` that have been encoded as defined in the previous subsection. The construction is performed recursively starting with the biggest feature f_n. First, the selection refinement rule with the cases that feature f_n is selected and deselected is applied. For the case that f_n is deselected, an `original` call is applied. For the case that it is selected, the refinement steps that implement method `m` in feature module f_n are inserted, which may contain further `original` calls that need to be resolved. Additionally, the refinement steps may contain variational specifications with original predicates that are encoded according to Definition 5.17 with the next smaller feature f_{n-1}. Next, all `original` calls are replaced by another selection refinement for the next smaller feature f_{n-1} followed by all steps that have been applied for feature f_n. This procedure is repeated recursively until the smallest feature f_0 is reached. For the case that the smallest feature is deselected, the skip refinement rule is applied. As the refinement steps of method `m` in the smallest feature are not allowed to contain further `original` calls, the encoding of the feature module implementations is finished.

We adapt the approach of Thüm et al. (2012) in two places. First, we always introduce the case where a feature f is not selected, because the selection refinement rule as defined in Definition 2.2, has the side condition that precondition `P` implies a disjunction of the guards, which in other words means that always one of the guards has to evaluate to `true`. Second, when inserting the refinement steps of a method in the case where feature f has been selected, we include non-encoded specifications (see step 2 in Definition 5.19). As these specifications can contain new uninterpreted original predicates and variable predicates, we need to resolve them by applying the variable predicate transformation in Definition 5.18, and the original predicate transformation as defined in Definition 5.17 with the next smaller feature f' that implements the currently encoded method.

To transform an `original` call, we define the following transformation operator.

Definition 5.19: Original Call Transformation (Bordis et al., 2022b)
Let the context of the `original` call be context($\mathtt{b} := \mathtt{original}(\mathtt{a}_1, ..., \mathtt{a}_n)$) $= (FM, f, \mathtt{m})$ and f' the next smaller feature according to the feature composition order from FM with $\mathtt{m} \in \mathrm{impl}(f')$.
Then $\{\mathtt{P}\}$ $\mathtt{b} := \mathtt{original}(\mathtt{a}_1, ..., \mathtt{a}_n)$ $\{\mathtt{Q}\}$ can be transformed in three steps:

1. *replace* $\mathtt{b} := \mathtt{original}(\mathtt{a}_1, ..., \mathtt{a}_n)$ *by* $\{\mathtt{P}\}$ if $f' \rightarrow \mathtt{S1}$ elif $\neg f' \rightarrow \mathtt{S2}$ $\{\mathtt{Q}\}$

2. *refine* `S1` *to* $\{\mathtt{P} \wedge f'\}$ $\mathtt{S1}'$ $\{\mathtt{Q}\}$ *where* $\mathtt{S1}'$ *is further refined using the refinement steps (including non-encoded specifications) of* `m` *in* $\mathrm{impl}(f')$

3. *refine* `S2` *to* $\{\mathtt{P} \wedge \neg f'\}$ $\mathtt{S2}'$ $\{\mathtt{Q}\}$ *where*
$$\mathtt{S2}' \textit{ is further refined to } \begin{cases} \mathrm{skip}, & \textit{if } f' \textit{ is the smallest feature impl. method } \mathtt{m} \\ \mathtt{b} := \mathtt{original}(\mathtt{a}_1, ..., \mathtt{a}_n), & \textit{otherwise} \end{cases}$$

```
M = (Limited → ((data^old.length < LIMIT) → (Base → (contains(data,
    newTop) & containsAll(data^old, data))))) & (!Limited → (Base → (
    contains(data, newTop) & containsAll(data^old, data))))
```

Listing 5.14: Variability encoded intermediate condition M of the metaproduct of method push (Bordis et al., 2022b).

Example 5.10. *For the IntegerList product line, we present the transformed method* push *in Figure 5.3 which has been constructed using the encoding of feature module implementations as described in Definition 5.19. The encoding starts with a Hoare triple* {P} S {Q} *with encoded pre- and postcondition as presented in Definition 5.17. The features that implement* push *are* Base, Limited, *and* Sorted. *As* Sorted *is the greatest feature in the feature composition order, the first refinement is a selection refinement with the cases that* Sorted *is selected and that it is not selected* ①. *For the case that* Sorted *is selected, the refinement steps of* push *in feature module* Sorted *are applied (composition* ②, original *call* ④ *(In Figure 5.3, the* original *call has already been resolved as described in the following.), and method call to* sort ⑤). *As refinement step* ② *is a composition, a new intermediate condition* M *is introduced. As intermediate condition* M *contains an* \original_post, *we need to eliminate the uninterpreted predicate. We show the result in Listing 5.14. Next, we continue to resolve the* original *call from the refinement steps of method* push *in feature* Sorted *by inserting the case distinction for feature* Limited ④. *At this point, we abbreviate the refinement steps of* push *in feature modules* Limited *and* Base *with the notation* [impl. of Limited/Base]. *For the case that* Sorted *is not selected, the next selection statement with cases* Limited *and* !Limited *is applied* ③. *Afterwards, the refinement steps of* push *in* Limited *are added for the case that* Limited *is selected, and another selection of feature* Base *is added for the case that feature* Limited *is deselcted. The refinement steps of method* push *in feature* Limited *may contain further* original *calls and specifications containing* original *predicates. These are encoded accordingly. These steps are repeated until feature* Base *is reached. As it is the smallest feature according to the feature composition order, there can be no further* original *calls in the refinement steps and* original *predicates in the specifications. Additionally, for the case that* Base *is deselected, a skip refinement is applied. As a result, there is no abstract statement,* original *call, or* original *predicate left in the metaproduct and the encoding of the feature module implementations for method* push *is complete.*

Encoding of the Feature Model

To encode the implementations in the feature modules and the specifications, we use boolean feature flags representing the selection or deselection of a feature. The combination of features to form valid product variants is restricted by the feature model. The feature model needs to be encoded so that the relevant assignments of the feature flags can be included in the metaproduct. Feature models are usually

Feature Model	**Propositional Formula**
Optional Feature C_i	$C_i \rightarrow P$
Mandatory Feature C_i	$(C_i \rightarrow P) \land (P \rightarrow C_i)$
Or-Group	$P \leftrightarrow \bigvee_{1 \leq i \leq n} C_i$
Alternative-Group	$(P \leftrightarrow \bigvee_{1 \leq i \leq n} C_i) \land \bigwedge_{i<j} (\neg C_i \lor \neg C_j)$

Table 5.1.: Translation of feature models into propositional formula (Thüm et al., 2011a). P is a parent feature with subfeatures $C_1, C_2, ..., C_n$.

```
FM = Root & ((Base → Root) & (Root → Base)) & (Limited → Root) & (
    Sorted → Root) & ((Sorted ↔ (Increasing | Decreasing)) & (!
    Decreasing | !Increasing))
```

Listing 5.15: Propositional formula of the *IntegerList* feature model (Bordis et al., 2022b).

presented in a tree structure as seen in Figure 2.3, but they can also be translated into a propositional formula using feature flags as atomic formulas (Batory, 2005). The propositional formula is constructed by conjoining the propositional formulas for each element of the feature model as shown in Table 5.1, the cross-tree constraints, and a formula selecting the root feature. For the feature model of the *IntegerList* product line, we show the propositional formula in Listing 5.15. For our metaproduct, we define the propositional formula of the feature model as a global condition.

Proof of Soundness

In this subsection, we reason about the soundness of our family-based verification strategy which means that the constructed metaproduct is correct for all valid feature configurations.

Theorem 5.3 (Soundness of the Family-Based Verification Strategy)**.** *Let* FM *be a feature model and* `m` *a method implemented by the product line. If the metaproduct constructed as described in Section 5.3.2 satisfies its specification, then all product variants for all valid feature configurations fc also satisfy their specifications.*

Proof. Hypothesis: Let $fc \in \text{FM}$ be a feature configuration, then the family-based encoding projected on feature configuration fc, $(\{\mathtt{P_{Meta}}\}\mathtt{Metaproduct}\{\mathtt{Q_{Meta}}\})|_{fc} \equiv \{\mathtt{P}_{fc}\}\mathtt{m}_{fc}\{\mathtt{Q}_{fc}\}$.

We show the above result by structural induction over the family-based encodings for the specification, feature modules, and feature model as defined in Section 5.3.2. The metaproduct projected onto feature configuration fc $(\{\mathtt{P_{Meta}}\}\mathtt{Metaproduct}\{\mathtt{Q_{Meta}}\})|_{fc}$ is defined as the assignment of all feature flags to `true` iff the corresponding features are contained in feature configuration fc and to `false` otherwise.

- *Specification*: The specification for the metaproduct is encoded by using the feature flags as antecedent and the specification defined in the feature module of this feature as consequent of an implication for original predicates (see Definition 5.17) and the presence conditions as antecedents and defined formulas as consequence for variable predicates (see Definition 5.18). Consequently, the implications of the metaspecification that have a feature flag that is not contained in feature configuration *fc* as antecedent are immediately evaluated to `true`. The remaining parts are implications with selected features in the antecedents. Therefore, the consequences all have to evaluate to `true` for this feature configuration to fulfill the specification and removing the implications such that only the consequences remain is equivalent. The resulting specification is now equivalent to the composed specification in the product-based generation over feature configuration *fc*.
- *Implementation*: The implementation of a method `m` is encoded by using the feature flags in selection refinement rules and afterwards inserting the refinement steps as defined in the feature module for that feature (see Definition 5.19). As the selection statements always just contain the two cases whether a feature is selected or not, a concrete path through the implementation can be determined. The refinement steps of this path are equal to composing the method refinements as defined by FeatureHouse (Apel et al., 2013b; Apel and Lengauer, 2008).
- *Feature Model*: As feature configuration *fc* is valid as defined in Definition 2.3, the variability encoded propositional formula evaluates to `true` for the feature flag assignment. The transformation of feature models to propositional formulas as shown in Section 5.3.2 has been proposed by Batory (2005) and is considered to be correct.

Combining the three components, a concrete CbC product $\{\mathtt{P}_{fc}\}\mathtt{m}_{fc}\{\mathtt{Q}_{fc}\}$ is formed which satisfies its specification because also the metaproduct does. □

5.3.3. Discussion

In this subsection, we discuss several observations we made during our work on the product-based and family-based verification strategies to guarantee their correctness. We address limitations regarding the combinatorial explosion of the sets of relevant feature sequences for the variational specifications and the product-based verification strategy. Additionally, we share our insights on the effects of applying post-hoc verification strategies on an incremental development approach like CbC and its classification in the classification scheme for verification strategies proposed by Thüm et al. (2014a).

Exponential Explosion Our proposed variational specifications and refinement rules for the product-based verification strategy generate the corresponding sets of relevant feature sequences and require the side conditions of all relevant method variants to guarantee the correctness of the product line. Therefore, our concept can be categorized as product-based analysis strategy, which means the analysis of each (method) variant in isolation (Thüm et al., 2014a), even if we technically do not verify complete product variants of the product line. We are aware of the fact that this does not scale to larger product lines due to a combinatorial explosion of possible product variants. To address this problem, we proposed a family-based verification strategy for product line analysis, which generates one variability encoded metaproduct that simulates the behavior of all product variants (Thüm et al., 2014a). For post-hoc verification, family-based verification strategies are more efficient in terms of needed proof nodes and verification time than product-based verification strategies (Thüm et al., 2014a; Thüm et al., 2012). We found the opposite case in our evaluation in Section 5.5: our product-based verification strategy outperformed the family-based verification strategy. However, we also identified further potential for improvement of the family-based verification strategy such that the proof efficiency may be increased in future work.

Application of Post-Hoc Verification Approaches to CbC In this chapter, we applied a family-based post-hoc verification technique to CbC. This family-based verification strategy generates a metaproduct simulating the behavior of every possible product in the product line *after* implementing and specifying the whole product line in a feature-based way. As the specification and verification in post-hoc verification is usually applied after implementation, this family-based verification strategy fits the development style of post-hoc verification well. However, one big advantage of CbC is that the implementation is developed incrementally together with the specification. Because of this incremental process, the correctness of a program developed with CbC can already be checked when the implementation of a program has not been finished yet. This advantage is lost through the post-hoc family-based verification strategy. Still, family-based verification strategies are popular for post-hoc verification, because they are more efficient in terms of proof nodes and verification time compared to other verification strategies for product lines (Thüm et al., 2014a; Thüm et al., 2012; Apel et al., 2013c).

Another two advantages of CbC compared to post-hoc verification are that errors can be tracked more easily and that the proof complexity decreases due to the fine-granular specification and refinement rules. These advantages are maintained by the family-based verification strategy and even increased, because the generated metaproducts become very large. For large methods, the proof complexity and the traceability of errors tends to become worse for post-hoc verification.

Generally, the family-based verification strategy is expensive for product lines that are under constant evolution, because the metaproduct has to be generated and completely verified again every time after the specification or implementation have been changed. This applies to both, CbC and post-hoc verification.

Classification of Verification Approaches Thüm et al. (2014a) introduced a classification schema for specifications and analyses techniques for software product lines. We classify our specifications as *feature-based* because the specification is applied in the feature modules without direct references to other features. Therefore, the specifications can be reused and directly describe the behavior of a feature.

We proposed two different verification strategies to check the correctness of the side conditions of the applied variational refinement rules of a product line based on a variational specification. The first is categorized as a product-based verification strategy, as it generates concrete side conditions for each relevant method variant whenever a variation point is reached. However, a product-based verification strategy as defined in the classification (Thüm et al., 2014a) generates every possible product variant and verifies its correctness. For example, methods that are not variable are checked every time they occur in a product variant, even though they stay the same. This makes these strategies inefficient. Our product-based verification strategy does not verify the same method multiple times. In fact, even in variable methods only the parts that contain variability (i.e., `original` calls, variational method calls, variable predicates, or original predicates) are checked for every relevant product variant. Consequently, we classify this verification strategy as *optimized product-based verification strategy*.

We classify the second verification strategy that is proposed in this chapter as *family-based analysis strategy* as all feature-based CbC methods are merged into one metaproduct. The variational specification is transformed into a family-based specification using feature flags to specify the behavior of all features in one. Additionally, the feature model is translated and added to the specification as well.

Formal Specification and Verification As with all formal verification techniques, our concepts strongly depend on the quality of the formal specification given by the developer. There is a correlation between the precision of a specification and the strength of the guarantees that the proof can give. The weaker the specification, the weaker is also the guarantee provided by the proof and vice versa. Additionally, we can only guarantee the correctness of the program with respect to its specification and not the correctness of the specification itself.

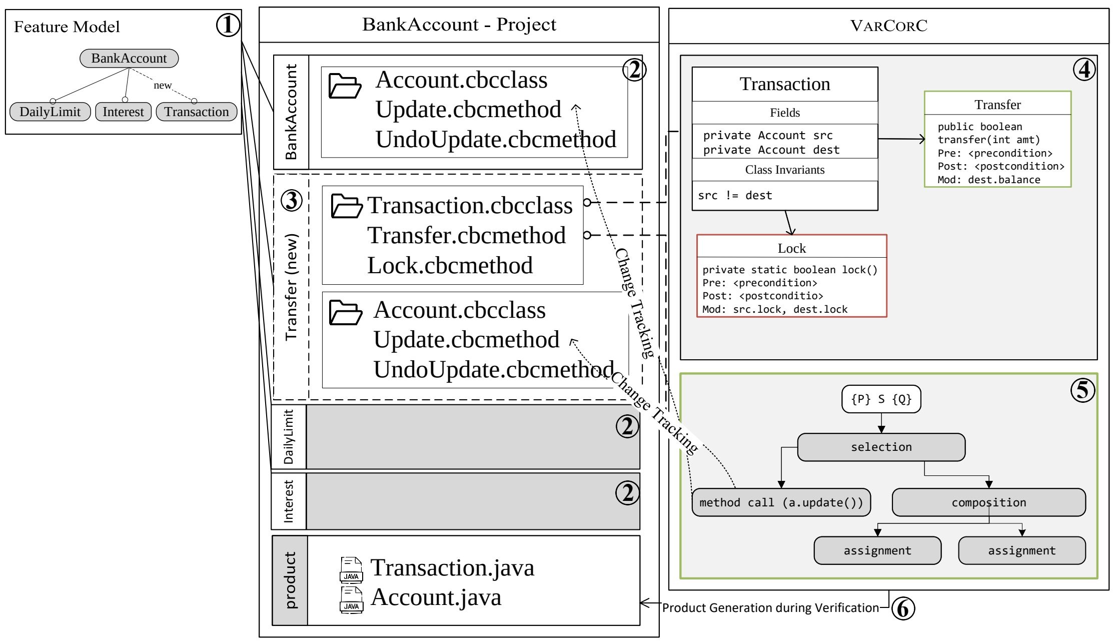


Figure 5.5.: Development process for object-oriented software product lines in VarCorC (Bordis et al., 2023a).

5.4. VarCorC: Tool Support for Object-Oriented Software Product Lines using Correctness-by-Construction

In this section, we present tool support for the development of software product lines using variational CbC as proposed in this chapter. The concept is implemented in an extension of CorC, which we call VarCorC. In VarCorC, we support the development of object-oriented software product lines, which combines the concepts presented in Chapter 3 and in this chapter. In this section, we first describe the development process as a scenario where a new feature in the product line is added. Second, we give implementation details of VarCorC. Third, we discuss limitations of the object-oriented approach in the tool VarCorC.

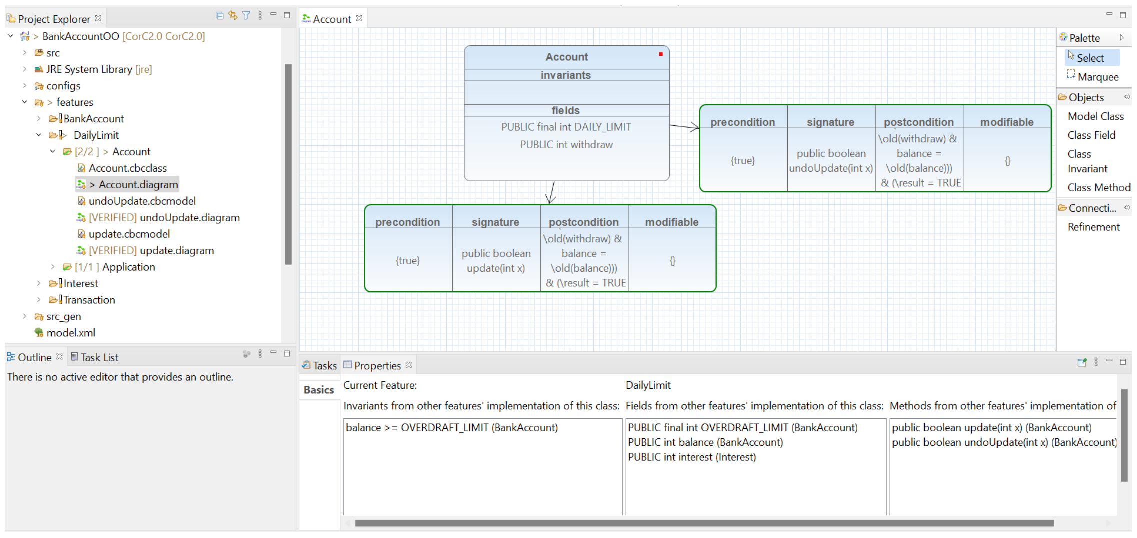
The Development Process with VarCorC

In the following, we describe the development process in VarCorC as shown in Figure 5.5 from the perspective of developer Alice. Alice develops an SPL that implements a bank account system and has already created a feature model ①. For the *BankAccount* SPL, Alice defined the features *BankAccount* (provides a base implementation of an account), *DailyLimit* (adds a limit that can be withdrawn from the account per day), and *Interest* (adds an interest to the account). Apart from the root feature *BankAccount*, all of the features are optional, which means that the user can select them individually. Alice already implemented the features *BankAccount*, *Interest*, and *DailyLimit* in separate feature modules ②. A feature can add new classes or extend already existing classes by adding fields, class invariants, and methods or by refining existing methods using the concepts introduced in Sections 2.1 and 5.2.

Alice now wants to add a feature *Transaction* to enable a transfer of money between accounts ③. Therefore, she inserts a new class called `Transaction` which is displayed as UML-like class diagram in VarCorC ④. She defines the fields `src` and `dest` of type `Account`, a class invariant, and methods `transfer` and `lock` to transfer money between two accounts and to lock an account such that the balance is unmodifiable. She defines these two methods with a signature and a method contract consisting of a first-order logic pre- and postcondition. Afterwards, she implements method `transfer` in the corresponding cbcmethod file ⑤ starting with the defined pre- and postcondition from the method contract. For the implementation, she uses the basic set of CbC refinement rules as defined in Definition 2.2 and our refinement rules for method calls and `original` calls from Definitions 5.8 and 5.10. For each applied refinement rule, the side condition is checked in the background by generating a proof file which is (semi)-automatically proven by the program verifier KeY (Ahrendt et al., 2016a) (more details in Section 2.2.2). Therefore, the method under development is guaranteed to be correct.

When verifying one of the four variation points, either the product-based verification strategy (Section 5.3.1) or the family-based verification strategy (Section 5.3.2) needs to be selected. In case of the product-based verification strategy, all relevant feature configurations of a method according to the feature model are generated into Java classes ⑥. For the family-based verification strategy, a metaproduct is generated. In each metaclass, the corresponding fields, class invariants, and metamethods are defined. The encoding of metamethods is done by using the verification strategy described in Section 5.3.2. Afterwards, each metamethod can be checked regarding correctness.

VarCorC's main usability features provide Alice with an overview on the verification status of all methods in the SPL and the traceability of errors down to one refinement step. The first is the class view where Alice can see the verification status of all methods of a class with red and green borders. The latter is naturally supported by the refinement-based approach of CbC and displayed with red and green borders for refinement steps in cbcmethods. Additionally, Alice is notified by a *change tracking mechanism* that updates the verification status of single refinement steps that depend on the contract of other methods, such as method calls and `original` calls. For example, Alice calls method `update` to implement method `transfer` from class `Account` ⑤. To guarantee the correctness of this refinement step, the specification of method `update` is checked to comply with the specification used in this refinement step. However, if Alice changes the specification of method `update` in another feature, the refinement step in method `transfer` has to be re-verified. VarCorC checks for these dependencies in the background and marks corresponding verification steps as "not verified" and notifies Alice about affected parts.



```
> BankAccountOO [CorC2.0 CorC2.0]
  src
  JRE System Library [jre]
  configs
  > features
    BankAccount
    DailyLimit
      [2/2 ] > Account
        Account.cbcclass
        > Account.diagram
        undoUpdate.cbcmodel
        [VERIFIED] undoUpdate.diagram
        update.cbcmodel
        [VERIFIED] update.diagram
      [1/1 ] Application
    Interest
    Transaction
  src_gen
  model.xml
```

There is no active editor that provides an outline.

Account
invariants
fields
PUBLIC final int DAILY_LIMIT PUBLIC int withdraw

precondition	signature	postcondition	modifiable
{true}	public boolean undoUpdate(int x)	\old(withdraw) & balance = \old(balance))) & (\result = TRUE	{}

precondition	signature	postcondition	modifiable
{true}	public boolean update(int x)	\old(withdraw) & balance = \old(balance))) & (\result = TRUE	{}

Current Feature: DailyLimit

Invariants from other features' implementation of this class:	Fields from other features' implementation of this class:	Methods from other features' implementation of
balance >= OVERDRAFT_LIMIT (BankAccount)	PUBLIC final int OVERDRAFT_LIMIT (BankAccount) PUBLIC int balance (BankAccount) PUBLIC int interest (Interest)	public boolean update(int x) (BankAccount) public boolean undoUpdate(int x) (BankAccount)

Figure 5.6.: Screenshot of VarCorC with class `Account` in feature *DailyLimit* (Bordis et al., 2023a).

Implementation Details

In the following, we give implementation details for our tool VarCorC[3] supporting the development of object-oriented SPLs using CbC and feature-oriented programming. VarCorC captures the CbC structure of methods and classes through a meta-model modeled with Eclipse Modeling Framework.[4] The graphical editor visualizes instances of the underlying meta-model in a tree-like structure for methods and UML-like class diagrams.

In Figure 5.6, we show a screenshot of VarCorC with class `Account` in feature *DailyLimit*. The project structure consists of a feature model, feature modules, and class folders. The cbcclass and cbcmethod files are split into a $\langle methodName \rangle \backslash \langle classname \rangle .diagram$ file, which contains the graphical information, and a $\langle methodName \rangle .cbcmodel \backslash \langle classname \rangle .cbcclass$ file, which is an instance of the corresponding meta-model. The *src-gen* folder contains generated Java classes, which store composed product variants for the product-based proofs.

In the bottom properties view, we show SPL information, such as all valid feature configurations or accessible fields and methods. The information displayed differs for classes and methods. In this case, we provide an overview of class invariants, fields, and methods of class `Account` in other features.

In the center of Figure 5.6, class `Account` in feature *DailyLimit* adds two fields (`DAILY_LIMIT` and `withdraw`) and two methods (`update` and `undoUpdate`). As displayed in the properties view, both methods have already been defined for this class

[3] VarCorC implements SPL development using CbC and is part of the tool CorC: https://github.com/KIT-TVA/CorC

[4] https://eclipse.dev/emf/

in feature *BankAccount*, which means that they are refined and can use an `original` call to call their implementation in feature *BankAccount*.

To guarantee the correctness of a whole SPL, we proposed two verification strategies. For the product-based verification strategy, we generate concrete side conditions of a refinement step according to the set of relevant feature sequences for the variation points used in the refinement step (e.g., an `original` call refinement rule with original predicates in its specification). To generate concrete side conditions, first the set of relevant feature configurations is calculated using the FeatureIDE library (Meinicke et al., 2017). We provide necessary information, such as the feature model and the information that the context of the refinement step (its feature and method) provides. For each feature configuration in this set of relevant feature configurations, we then resolve the variation points in a second step. Original predicates are resolved by composing the specifications for the feature configuration. For variable predicates, the presence conditions are resolved and the definition for the fulfilled presence condition(s) is defined in a KeY-file, such that KeY can interpret variable predicates during the proof. To resolve `original` and variational method calls, a partial product variant in the form of Java classes with composed contracts is generated. The `original` call in the KeY proof file is replaced by a call to the method stub in that generated product variant. When all variation points are resolved, KeY can prove whether the side condition of the refinement step holds for this particular feature configuration. The second step is repeated for all feature configurations in the set of relevant feature configurations calculated in the first step. If all these proofs are successful, the statement is considered to be correct.

For the family-based verification strategy, a metaproduct is generated in a separate meta-folder. This generation of the metaproduct from variational CbC implementations in the individual feature modules is performed solely in VarCorC. We generate a class with static constants for the features that can be accessed globally. For each class, we generate a new cbcclass that contains all fields, class invariants, and methods that have been introduced in the feature modules. For each method, we create a new cbcmethod diagram starting with a Hoare triple and the metapsecification contract. Then, we start to recursively insert refinement steps of the individual feature modules in the order defined by the feature model resolving any variation point as described in Section 5.3.2. The feature model is translated into a propositional logic formula and provided as a global condition in the metamethod. Once the generation of the metaproduct is completed, it can be be verified using the original set of refinement rules in Definition 2.2 using KeY as described in Section 2.2.2 because no variational constructs are contained in metamethods.

Discussion about Reachability of Variables

In the following, we discuss a limitation regarding syntactical correctness that is a result of adding classes with fields in VarCorC and the proposed verification strategies. In the implementation of VarCorC, we support the development of object-oriented software product lines. As explained above, features can add fields to a class. For the product-based verification strategy, products are generated according to the set of relevant feature sequences. Thereby, the fields that every feature in the feature configuration adds to a certain class are generated into that class. For the family-based verification strategy, the fields in a class are simply collected and given in the implementation of the metaclass. One important guarantee that we cannot give following our verification strategies is that all product variants in the product line are syntactically well-formed in the sense that all fields used in the implementation and specification of a feature are reachable in every valid product variant of the product line. To check this, we would have to perform our product-based verification strategy for all product variants, which is not feasible due to the exponential explosion of the number of product variants. With the family-based verification strategy, this is not checkable as of now. Usually, this is one of the tasks that is statically checked by a type system before product variants are generated in a product line. As CorC was developed as a tool to support the CbC development approach and not to actually statically analyze code, its architecture is not easily adaptable such that checks like this can be integrated. For now, this remains a limitation of the tool, but we plan to include well-formedness checks of programs in future work. For example, since delta-oriented programming (Schaefer et al., 2010) shares similar concepts with feature-oriented programming, such as modular feature implementations that are composed and a similar code reuse mechanism, we could adapt Schaefer et al. (2011)'s modular, constraint-based type system for delta-oriented programming.

5.5. Evaluation

In this chapter, we proposed different concepts for the development of provably correct software product lines using CbC. In particular, we proposed two concepts for specifying software product lines (original and variable predicates), two CbC refinement rules for the variation points in the code of a software product line (`original` and variational method calls), two verification strategies to show the correctness of the two new refinement rules (a product-based and a family-based verification strategy), and our tool VarCorC, which implements all these concepts with the addition of object-orientation. In this section, we evaluate the feasibility of all concepts, the specification effort and proof efficiency in comparison to post-hoc verification as implemented in KeY, and the proof efficiency of the family-based and product-based verification strategies. For the evaluation, we use VarCorC as presented in Section 5.4.

5.5.1. Research Questions and Subject Systems

We are investigating four research objectives. Research question **RQ1** gives us insights into the extent to which our proposed concepts, in particular the product-based and family-based verification strategies, variable predicates, and object-orientation as implemented in VarCorC, can be used to create software product lines using CbC. With research questions **RQ2** and **RQ3**, we compare our concepts to develop correct-by-construction software product lines with post-hoc verification in terms of proof efficiency and specification effort. By answering research question **RQ4**, we analyze which of the proposed verification strategies is more efficient in terms of proof nodes. In particular, we address the following research questions:

RQ1 (feasibility): Is it possible to develop correct-by-construction software product lines using original predicates and our concept for ...

- **RQ1.1:** ... product-based verification?
- **RQ1.2:** ... family-based verification?
- **RQ1.3:** ... variable predicates?
- **RQ1.4:** ... object-orientation in our tool VarCorC?

RQ2 (specification effort): How does the effort of specifying software product lines in VarCorC compare to post-hoc verification in KeY?

RQ3 (external proof efficiency): How does the number of proof nodes needed for verifying software product lines with the product-based verification strategy in VarCorC compare to post-hoc verification in KeY?

RQ4 (internal proof efficiency): How does the number of proof nodes in the family-based verification strategy compare to the product-based verification strategy?

We evaluate our concepts mainly on two prominent subject systems, namely *IntegerList* (Scholz et al., 2011), and *BankAccount* (Thüm et al., 2012). For the feasibility evaluation of object-orientation as implemented in VarCorC (research question **RQ1.4**) we additionally use a third subject system, the *Elevator* product line (Plath and Ryan, 2001), as the *Elevator* product line consists of more classes and methods than the *IntegerList* and *BankAccount* product lines. All of the subject systems were already used for specifying software product lines (Thüm et al., 2019).

In Table 5.2, we summarize some characteristics of the three subject systems. Besides standard characteristics for software product lines, such as the number of features, feature configurations, classes, and methods, we also include the number of variation points in the code. This metric gives us insight into how variable the implementation of the methods is. In our case, variation points can either be `original` call or variational method call refinement rule applications. For example, the method call refinement that calls method `sort` in Figure 5.2 is a variational method call refinement because the implementation of method `sort` depends on the selection of features *Increasing* and *Decreasing*. Lastly, we also show the number of variable predicates that we defined for each subject system to answer research question **RQ1.3**.

We already used the *IntegerList* product line as a running example throughout this thesis (see Sections 2.3 and 5.1 to 5.3). The product line is implemented with five features and three variation points in the code.

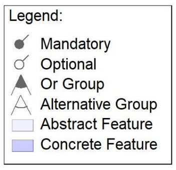
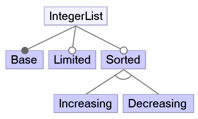
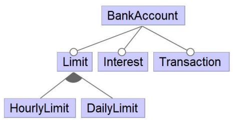
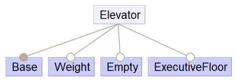
	IntegerList	*BankAccount*	*Elevator**
Feature model  Legend: Mandatory Optional Or Group Alternative Group Abstract Feature Concrete Feature	 IntegerList Base Limited Sorted Increasing Decreasing	 BankAccount Limit Interest Transaction HourlyLimit DailyLimit	
#Concrete Features	5	6	4
#Feature Configurations	6	16	8
#Classes	1	3	4
#Method Refinements in Feature Modules	5	13	38
#`original` calls	2	7	4
#Variational method calls	1	3	0
#Original Predicates	5	16	9
#Variable Predicates$^{\triangle}$	1	3	0

Table 5.2.: Overview of the subject systems *IntegerList*, *BankAccount*, and *Elevator*. The number of method refinements corresponds to the number of feature-oriented programming refinements/overrides. *The *Elevator* subject system has only been used for the feasibility evaluation of object-orientation as implemented in VarCorC (**RQ1.4**).$^{\triangle}$Variable predicates are only used in the feasibility evaluation (**RQ1.3**).

The *BankAccount* product line has been briefly introduced in Section 5.4. It implements basic functionality of a bank account, such as updating its balance and transferring money from one account to another. Furthermore, it supports the definition of hourly and daily limits for withdrawals. The *BankAccount* product line is implemented with six features (*BankAccount*, *Limit*, *DailyLimit*, *HourlyLimit*, *Interest*, and *Transaction*) and three classes (`Account`, `Application`, and `Transaction`) with 13 method refinements. For example, a method `update` is implemented in two feature modules, which means it has two method refinements. Feature *BankAccount* implements the basic functionality of the bank account in class `Account`. Feature *Limit* introduces a limit that is not allowed to be exceeded by method `update`. Depending on the features *HourlyLimit* and *DailyLimit*, a limit for dispenses in one hour or one day is introduced. It is also possible to include both types of limits.

The *Elevator* product line implements a passenger elevator that can move up and down. Persons can call the elevator on each floor and enter and leave the elevator. The *Elevator* is implemented with four features: *Base*, *Empty*, *ExecutiveFloor*, and *Weight*. The *Elevator* product line implements four classes called `Elevator`, `Environment`, `Floor`, and `Person`. It consists of 38 method refinements that are used to instantiate and manipulate object instances. The *Elevator* product line has four variation points in code.

5.5.2. RQ1 – Feasibility

The first research question **RQ1** focuses on feasibility and is answered by recreating the JAVA implementations of the subject systems with JML specifications in VARCORC and then checking their correctness. We divided this research question into four sub-research questions to investigate the different concepts in isolation. Thereby, the original predicates and the refinement rules for the `original` and variational method calls from Sections 5.2.1 and 5.2.2 build the basis. Their feasibility is evaluated in the first two sub-research questions (**RQ1.1 and RQ1.2**) by checking their correctness either with the product-based or the family-based verification strategy. For both strategies, we checked that all refinement steps, 1) using contract composition over the set of relevant feature sequences for the product-based verification strategy and 2) in the metamethods for the family-based verification strategy, are verifiable. For research question **RQ1.3**, we modified the specifications of each subject system by adding variable predicates. For all three subject systems, we checked whether VARCORC was still able to prove the correctness with the defined variable predicates. In research question **RQ1.4**, we transferred the subject systems into the object-oriented structure as introduced in Section 5.4 and checked whether the refinement steps could still be verified. For research questions **RQ1.1 – 1.3**, we used the *IntegerList* and *BankAccount* product lines. For research question **RQ1.4**, we additionally used the *Elevator* product line. We discuss the results in the following.

RQ 1.1: Product-based Verification

The product-based verification strategy generates a set of relevant feature sequences for each variation point (original predicates in the specification and `original` calls and variational method calls in the code) and composes the contracts for each feature sequence to guarantee the correctness for all valid product variants. In total, we could derive 22 different method variants with respect to valid feature configurations of the feature models. Thereby, six of the method variants are in the *IntegerList* product line (method `push` has four variants, and method `sort` has two), and 16 in the *BankAccount* product line (methods `update`, `undoUpdate`, `transfer` and `nextYear` have two variants each, and `nextDay` and `nextHour` have four). This includes nine `original` calls, four variational method calls, and 21 original predicates. As expected, all proofs passed for all method variants. We conclude that with VARCORC, it is possible to develop correct-by-construction software product lines using our product-based verification strategy.

RQ1.2.: Family-based Verification

We used VARCORC to generate the metaproduct for each subject system by encoding the variability of the feature modules, the specification, and the feature model for each method (see Section 5.3.2). In total, we could guarantee the correctness of all product

$$\texttt{limitExceeded(int withdrawD, int withdrawH, int limitD, intlimitH)} = \begin{cases} (\texttt{withdrawD} >= \texttt{limitD}) & \textit{iff} \quad \textit{DailyLimit} \land \textit{Limit} \\ ((\texttt{withdrawH} >= \texttt{limitH}) & \textit{iff} \quad \textit{HourlyLimit} \land \textit{Limit} \end{cases}$$

Listing 5.16: Defintion of variable predicate `limitExceeded`.

variants by verifying all CbC diagram nodes of the seven metamethods (methods `push` and `sort` from the *IntegerList* product line and methods `update`, `undoUpdate`, `transfer`, `nextDay`, and `nextYear` from the *BankAccount* product line). As expected, all proofs passed for all metaproducts. We conclude that with VarCorC, it is possible to develop correct-by-construction software product lines using our family-based verification strategy.

RQ1.3: Variable Predicates

The focus of variable predicates is to increase the precision of software product line specifications, which does not have a quantitative metric that can be measured. We already discussed the trade-off between precision of a specification and increased mental effort in Section 5.2.3. Therefore, we concentrate on a feasibility evaluation where we add variable predicates to the three subject systems. For the *IntegerList* product line, we introduced the variable predicate `isSorted` to precisely specify the order in which the call of method `sort` sorts the array (see Listing 5.3). For the *BankAccount* product line, we added three variable predicates. First, the variable predicate `balanceChangedResult` checks whether the balance of a provided account changed by a given amount in the course of a method. Second, the variable predicate `transferPerformed` checks whether a transfer between two given accounts by some amount was successful. Third, the variable predicate `limitExceeded` decides whether a defined limit is exceeded.

To showcase another variable predicate, we present variable predicate `limitExceeded` from the *BankAccount* product line in Listing 5.16. The limits are defined in features *HourlyLimit* and *DailyLimit* in an or group in the feature model. Therefore, the variable predicate is defined with two definitions. The first one checks whether the daily limit `limitD` is exceeded by the withdrawal in case that feature *DailyLimit* has been selected. The second one does the same for the hourly limit `limitH` in case that feature *HourlyLimit* has been selected. These two cases are sufficient to precisely specify all products containing one of the limits and both limits because in case that both presence conditions are fulfilled, the definitions are conjoined (see Section 5.2.1). Variable predicate `limitExceeded` is used in the specification and in an if guard to implement the behavior of method `update` in feature *Limit*. Without using a variable predicate, the definitions for these two cases would need to be connected with a disjunction (making it less precise) or expressed with feature flags that need to be set when a feature configuration is checked (as precise as using variable predicates, but more effort for the user). Consequently, the use of this variable predicate does not only increase the precision, but it also increases readability and maintainability compared to specifying cases with disjunctions and using feature flags. This effect even increases when we want to add more limits to the or-group of the feature model.

In total, we defined four variable predicates for the two subject systems that we conducted. In all of these cases, we made the specifications more precise by removing disjunctions where the cases could be made explicit with presence conditions in variable predicates. Since the specification changed in these four cases, we successfully re-verified all subject systems with VarCorC to ensure that the specification is still correct. We conclude that with VarCorC, it is possible to develop correct-by-construction software product lines using variable predicates.

RQ1.4: Object-orientation in VarCorC

VarCorC (see Section 5.4) supports the concepts as described in this chapter and includes object-orientation based on the concepts proposed in Chapter 3. For every class, we created cbcclass files and transferred all fields and class invariants from the original Java software product lines that we used as subject systems. In the prior sub-research questions, the class invariants were part of the initial pre- and postconditions of the methods in VarCorC. For this evaluation in **RQ1.4**, we removed them from the initial pre- and postconditions and declared them as class invariants in the cbcclasses. To show feasibility, we verified every refinement step in all methods individually for all valid feature configurations in VarCorC, therefore showing correctness of the whole software product lines. We conclude that with VarCorC, it is possible to develop object-oriented software product lines using CbC.

Answer to Research Question RQ1: Since we could answer all sub-research questions of research question **RQ1** positively, we conclude that it is possible to develop correct-by-construction software product lines using original predicates, product-based verification, family-based verification, variable predicates, and object-orientation as implemented in VarCorC.

5.5.3. RQ2 – Specification Effort

With this research question, we want to assess the specification effort of developing correct software product lines in CbC compared to post-hoc verification. We do this with the goal to put specification effort and proof efficiency (as discussed in research question **RQ3**) into relation. To enable the comparison to post-hoc verification, each method of our subject systems is implemented as a VarCorC program and a Java program annotated with JML, which is verified with KeY. The specification effort is

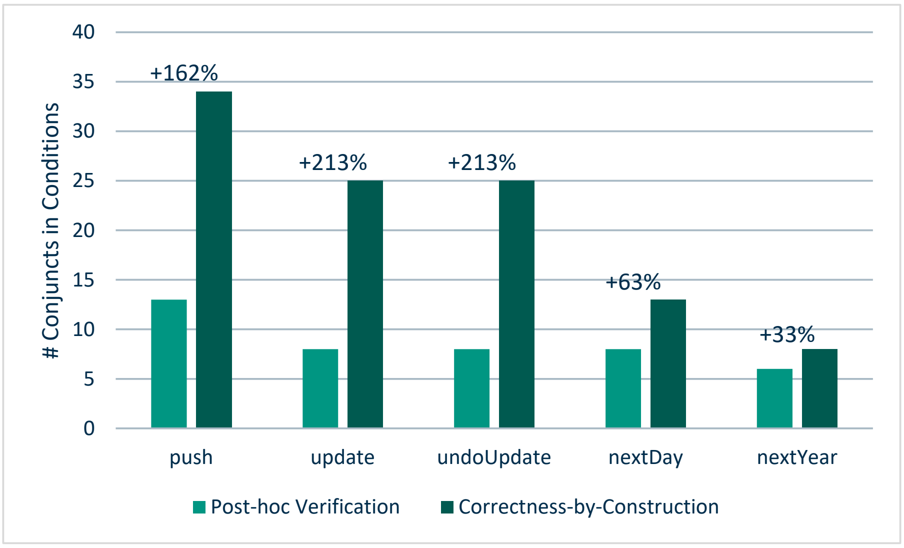


Figure 5.7.: Total number of conjuncts in conditions per method in the subject systems *IntegerList* and *BankAccount* (Bordis et al., 2020a).

measured by counting the number of conditions that have been connected using a conjunction in all user-provided specifications. This includes the pre- and postconditions, class invariants, and additionally the intermediate conditions, which only occur in VarCorC. Loop invariants are identical in both cases and are therefore omitted. For this research question, we do not use variable predicates in the specifications, because post-hoc verification in KeY does not support this concept.

Results and Discussion

In Figure 5.7, we illustrate the total specification costs for VarCorC and JML per method refinement with respect to precondition, postcondition, intermediate condition (only VarCorC), global conditions (only VarCorC), and class invariants (only JML). We found that the average number of conjuncts in conditions used for VarCorC were 162% higher for *IntegerList* and 130% for *BankAccount* than for Java and JML. However, these metrics do not fully reflect the complexity of the single conditions, as the conjuncts themselves range from rather complex (e.g., using an `\exists` clause) to quite simple (e.g., assuming the value of a variable to be zero). Nevertheless, even simple conditions, such as an object to be non-null, mean effort, as they still have to be specified, even if the manual effort is lower compared to more complex conditions. We also found that 58% of the additional conditions have been introduced by intermediate conditions, which tend to be more complex than the global conditions. However, our experience was that also most of the intermediate conditions did not take too much effort to specify, as CbC is applied on the fine-grained level of statements. Furthermore, when the postcondition is already known, the intermediate conditions often build up slowly and partially reflect the postcondition.

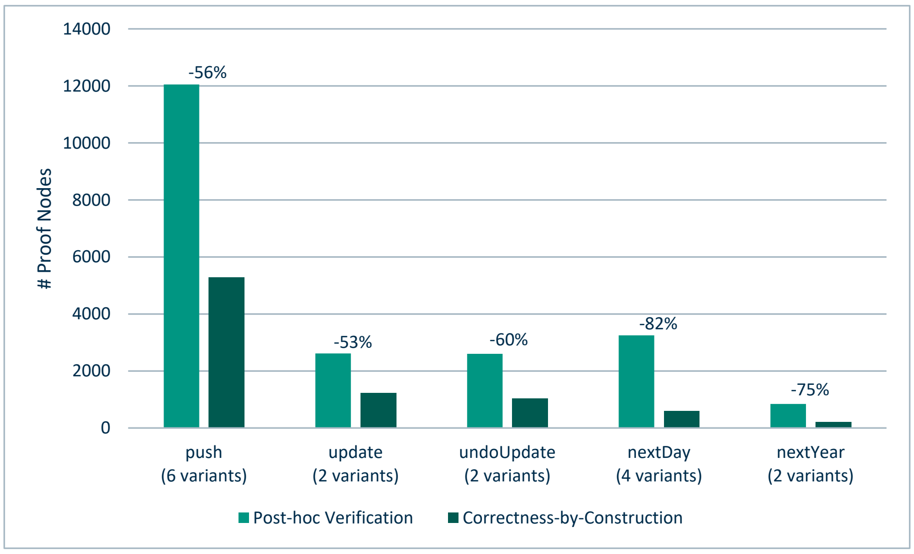


Figure 5.8.: Total number of proof nodes per method in the subject systems *IntegerList* and *BankAccount* (Bordis et al., 2020a). Only methods with `original` calls are evaluated. For both approaches, CbC and post-hoc verification, a product-based verification strategy was used.

Answer to Research Question RQ2: In summary, the specification effort using CbC as implemented in VarCorC is higher than using post-hoc verification with KeY, but we rate the complexity of the additional specifications as low.

5.5.4. RQ3 – External Verification Effort

To answer research question **RQ3**, where we want to assess the proof efficiency of developing correct software product lines in CbC compared to post-hoc verification with KeY, we verify each method of our subject systems, the *IntegerList* and *BankAccount* software product lines, using a product-based verification strategy. For CbC, we used the product-based verification strategy as defined in Section 5.3.1 and summed up the number of proof nodes that KeY needed for each method variant of each refinement step. For post-hoc verification, we manually generated the same method variants that have been verified with CbC and verified the whole Java method variants with KeY. We compare the proof efficiency regarding the total number of proof nodes that are needed to close the proofs. We collected the number of proof nodes as an indicator of the proof efficiency for every verified method variant for both approaches, CbC and post-hoc verification. For this research question, we do not use variable predicates in the specification, because post-hoc verification in KeY does not support this concept. The accumulated results for each method are shown in Figure 5.8.

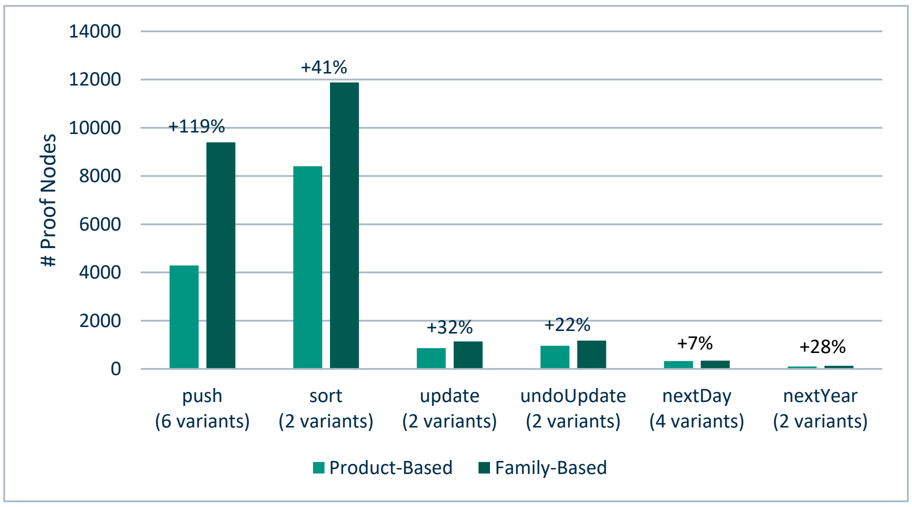


Figure 5.9.: Total number of proof nodes of the product-based and the family-based verification strategy in VarCorC (Bordis et al., 2022b). The results are shown per method in the subject systems *IntegerList* and *BankAccount*.

Results and Discussion

The verification with VarCorC needs between 53% and 82% fewer proof nodes than the verification with Java and JML. This result is in alignment with our expectations and previous evaluations on CorC with isolated algorithms (Runge et al., 2019) by indicating a reduced proof complexity, as the verification of one method is split into smaller sub-proofs using CbC. This means that even for smaller subject systems, the total number of proof nodes needed for VarCorC is at least twice as small as the number of proof nodes needed for post-hoc verification, and therefore, proofs with CbC seem to be more efficient. The effects may be considerably larger with more complex and varied subject systems.

Answer to Research Question RQ3: In summary, the fine-grained specifications of CbC seem to lead to a higher proof efficiency by lowering the number of proof nodes that the verification tool has to do. In combination with the results of research question **RQ2**, there is a trade-off in VarCorC between a higher specification effort and a higher proof efficiency.

5.5.5. RQ4 – Internal Verification Effort

With research question **RQ4**, we want to assess the proof efficiency of our two proposed verification strategies, the product-based and the family-based verification strategies. Similar to research question **RQ3**, we use the number of proof nodes that the deductive verifier KeY needs to close a proof as a metric for the proof efficiency. We evaluate this reserach question with the *IntegerList* and *BankAccount* product lines. For the product-based verification strategy, we verified the side conditions of

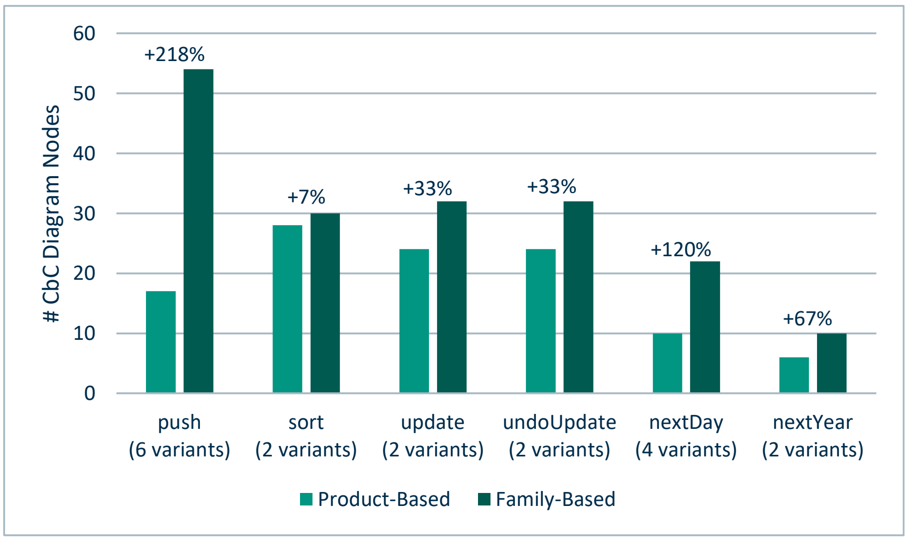


Figure 5.10.: Total number of CbC diagram nodes of the product-based and the family-based verification strategy in VarCorC (Bordis et al., 2022b). The results are shown per method in the subject systems *IntegerList* and *BankAccount*.

each refinement step in the feature-based CbC phase by composing the contracts over the set of relevant feature sequences (see Section 5.3.1). To get the total number of proof nodes per method, we summed up the needed number of proof nodes from these individual proofs of the refinement steps. For the family-based verification strategy, we encoded the product line into its metaproduct consisting of one metamethod per method as described in Section 5.3.2 and checked the side conditions of each refinement step in the metamethod. Thereby, we collected the proof nodes that KeY needed to close a proof. We present the total number of proof nodes per method for the family-based and the product-based verification strategy in Figure 5.9. To analyze the correlation between the number of proof nodes and the size of the CbC diagrams, we present the number of CbC diagram nodes per method in Figure 5.10. The number of CbC diagram nodes can be retrieved by counting the nodes in a CbC diagram in VarCorC.

Results and Discussion

For all methods, the family-based verification strategy needed more proof nodes compared to the product-based verification strategy, which means a decrease in proof efficiency and conseuqently an increase in proof complexity for the family-based verification strategy. The difference between the two verification strategies varies between 7% and 119% of proof nodes. To analyze the correlation between the number of proof nodes and the number of CbC diagram nodes, we additionally present the number of CbC diagram nodes per method in Figure 5.10. Thereby, the increase in the number of CbC diagram nodes for the family-based verification strategy does not always seem to correlate with an increase in the number of proof nodes. But, the increase of proof nodes for the family-based verification strategy seems to correlate with an already high number of proof nodes for the product-based verification strategy (see methods `push` and `sort` vs. `update`, `undoupdate`, `nextDay`, and `nextYear`). It is generally not surprising that the methods `push` and `sort` have the highest number of proof nodes, as they work on arrays and use multiple loops. These are constructs that are generally more complex to prove.

As method `push` has the biggest increase of proof nodes for the family-based verification strategy (+119%), we examine it further. Method `push` is refined three times, and therefore, three feature modules have to be encoded for the metaproduct. Additionally, the implementation of method `push` uses two `original` calls, which have to be resolved as well. This leads to an insertion of the refinement steps in feature *Base* (the smallest feature in the composition order) at four different places during the encoding of the feature modules. Consequently, the same refinement steps are verified four times for different cases of feature selections in the specification. As there are many refinement steps in feature *Base* also containing loops, this explains the high increase in the number of proof nodes (see Figure 5.9) and also CbC diagram nodes (see Figure 5.10) for method `push`. In contrast, method `nextDay` also has three refinements and two `original` calls. However, the increase of proof nodes has been the lowest (+7%), and also the number of CbC diagram nodes has not been as high as the one of method `push`. This is due to the fact that the CbC refinement steps of method `nextDay` in the smallest feature according to the feature composition order are fewer and also not complex to prove.

Method `sort` has the second-highest increase in proof nodes in the family-based verification strategy (+41%). However, in contrast to methods `push` and `nextDay`, it only has two feature refinements and also no variation points, because the features implementing it are in an alternative group. Therefore, the encoded metaproduct does not contain duplicate refinement steps and therefore only has two CbC diagram nodes more than the two feature-based method refinements in isolation (see Figure 5.10). The two additional CbC diagram nodes are two selection statements and a skip statement that are introduced during the encoding of the feature module implementations minus one because there is only one starting Hoare triple. There are two possible explanations for the high increase in the number of needed proof nodes. The first is that the side conditions of the additional CbC diagram nodes are complex to prove, and the second one is that the encoded specification of the family-based encoding is complex to prove. However, the added selection and skip statements usually are not complex enough to cause the difference. To validate this assumption, we verified the added refinements in isolation and found that they only add 199 proof nodes. Consequently, the encoded specification of the family-based verification strategy seems to make the proofs more complex. A possible explanation for this increase in comparison to the other methods is that method `sort` is implemented in features that require another feature to be selected (feature *Sorted*), which does not implement method `sort` itself. Therefore, the selection of *Sorted* is not directly encoded in the metaspecification and the feature modules, but only in the encoding of the feature model. Therefore, for every refinement step, the prover has to deal with the propositional formula of the feature model to verify the relationship between *Sorted* and its subfeatures.

Answer to Research Question RQ4: In our evaluation, the family-based verification strategy needed more proof nodes compared to the product-based verification strategy, and therefore, the product-based verification strategy has a higher proof efficiency. The main causes for the inefficiency of the family-based verification strategy were that 1) similar refinement steps need to be verified multiple times, which is due to the family-based encoding of original calls in the code, and 2) the metaspecification, including the encoding of the whole feature model, can lead to additional proof complexity.

5.5.6. Threats to Validity

In this section, we evaluated our concepts for correct-by-construction software product lines in terms of feasibility, specification effort and proof efficiency compared to post-hoc verification as implemented in KeY, and in terms of proof efficiency of the two proposed verification strategies. Even though we have conducted the evaluation with the greatest care, there are threats to the validity of the results that we discuss in the following.

External Validity. Our main concern with our evaluation is generalizability. The methods implemented in the three subject systems range in size from 1 to 30 lines of code with a maximum of three refinements through `original` calls in the code. Therefore, these subject systems do not reflect the scope of software product lines that are commonly used in practice. Consequently, general conclusions about the specification effort and proof efficiency between post-hoc verification and CbC (**RQ2** and **RQ3**) and the proof efficiency of our two verification strategies (**RQ4**) cannot be drawn from the evaluation. However, they do give us the opportunity to analyze each subject system in detail. Another assumption that we made was that the verification of the subject systems had to work automatically. In general, there are few open-source software product lines with formal specifications in the literature that meet this criterion.

Internal Validity. The evaluation was executed manually in large parts. We wrote most of the specifications for the three subject systems ourselves, and the proofs were triggered by hand. Even though the proofs run automatically, the results were taken manually from the proof reports of KeY. Thus, there may still be faults in the specifications or implementations, and we might have introduced transmission errors. All data was collected with utmost care and double-checked to exclude transmission errors. Since we were able to verify all of the methods with VarCorC and KeY, this is a strong indication of the correctness of the subject systems. In particular, we checked that the contracts of the methods for the two different approaches, CbC and post-hoc verification, were the same.

To evaluate the specification effort in research question **RQ2**, we used the number of conjuncts in specification conditions as a metric. This metric does not necessarily reflect the full complexity of specifications. Other metrics that could be used include formula length (e.g., number of characters), number of variables, and quantifier depth. However, for our subject systems, we found that formula length and the number of variables tend to oversimplify the complexity, and quantifiers were only used in the *IntegerList* subject system. Therefore, we chose the number of conjuncts as the metric that reflects the specification effort best for our subject systems. Nevertheless, we acknowledge this as an internal threat, as relying on a single metric cannot provide a complete analysis of the specification effort. For a deeper and more robust analysis, future work should combine multiple metrics to better capture the complexity of a specification to assess the effort of writing specifications for software product lines with CbC and JML.

5.6. Related Work

In this chapter, we addressed CbC for creating provably correct software product lines based on the software product line realization technique feature-oriented programming. Our contribution can be split into the feature-based CbC phase, where we proposed original predicates and variable predicates as specification techniques, and the verification phase, where we proposed an optimized product-based and a family-based verification strategy. In the literature, there are different concepts for specifying and verifying software product lines, which have been categorized by Thüm et al. (2014a). In the following, we present related work for software product line specification techniques that follow a design-by-contract scheme and software product line verification techniques in these categories. Both, related work on software product line specification and verification techniques, are dominated by post-hoc approaches.

5.6.1. Software Product Line Specification

Domain-Independent Specification. The most comprehensive and at the same time non-specific specification category is *domain-independent specifications* (Figueiredo and Cacho, 2008; Thüm et al., 2014a). This specification technique does not relate to a specific domain but defines rules that have to be fulfilled for every software product line implemented using a particular product line realization technique, e.g., feature-oriented programming (Prehofer, 1997). Parsers that check syntax conformance (Kästner et al., 2011), path coverage (Shi et al., 2012), or analyses showing the absence of dead code in product lines (Tartler et al., 2011) are other prominent examples using domain-independent specifications.

Family-Wide Specification. All product variants of a product line need to comply with a family-wide specification. These specifications are also called *global specifications*. As an example, in a product line for pacemakers, every product variant has to monitor and eventually generate a patient's heart beat (Liu et al., 2007). This specification technique is very restrictive as it cannot express varying behavior in the product variants.

Product-Based Specification. Theoretically, it is possible to specify every product variant from a product line individually using only formal specification techniques for regular software systems, such as design-by-contract for object-oriented programming (Meyer, 1988; America, 1991; Findler et al., 2001; Dhara and Leavens, 1996; Hatcliff et al., 2012). This technique of specifying every product variant on its own is referred to as *product-based specification* (Thüm et al., 2014a; Van Gurp et al., 2001). However, product-based specifications only scale for product lines with a low number of product variants and product lines whose product variants are largely disjunct. Additionally, defining and maintaining behavior that is the same for several product variants involves higher effort since adaptations have to be maintained in each product variant in isolation.

Feature-Based Specification. Another way to specify a product line is to specify the functionality of each feature in isolation (Apel et al., 2013c). To verify properties across features, feature-based specifications of selected features can be composed. In contrast to family-wide and product-based specifications, feature-based specifications offer the possibility to reuse a specification over several products, as a feature's specification is part of every product containing the corresponding feature. Consequently, compared to product-based specifications, the effort of specifying a product line decreases, and specifications become more maintainable. Because of these advantages, feature-based specifications are commonly used for compositional software product line realization techniques, as we have seen in our running example.

For *feature-oriented programming*, initial ideas used behavioral subtyping for feature-based method specifications (Batory et al., 2000), however, Agostinho et al. (2008) and Smaragdakis and Batory (2002) argue that this is too restrictive. In Section 5.2.1, we already mentioned that Thüm et al. (2019) investigated six contract composition techniques for feature-oriented programming. We adapted explicit contracting for CbC in our concept of the original predicates (see Section 5.2.1), which requires more fine-grained specifications than one pre- and postcondition per method.

For other composition-based product line realization techniques, such as *delta-oriented programming* and *aspect-oriented programming*, there are also several approaches for contract composition that are similar to the ones defined for feature-oriented programming (Ahrendt et al., 2014; Damiani et al., 2012; Hähnle et al., 2013; Clifton, 2005; Molderez and Janssens, 2015; Molderez and Janssens, 2012; Plath and Ryan, 2001; Agostinho et al., 2008). For example, Ahrendt et al. (2014), Damiani et al.

(2012), and Hähnle and Schaefer (2012) each propose similar contract composition techniques for delta-oriented programming to the ones proposed for feature-oriented programming, allowing to override, reuse, and refine method contracts following Liskov's substitution principle. Hähnle et al. (2013) propose a contract composition technique similar to explicit contracting for feature-oriented programming, which can additionally remove preconditions, postconditions, and frames. Other work focuses on defining specifications such that product lines can be analyzed more efficiently (Scaletta et al., 2021).

Each of the proposed specification techniques is tailored to its realization technique. Some of them are based on a substitution principle (Ahrendt et al., 2014; Damiani et al., 2012; Hähnle and Schaefer, 2012; Agostinho et al., 2008; Molderez and Janssens, 2012), while others transfer means from the realization technique to the specification level (Molderez and Janssens, 2015; Hähnle et al., 2013).

Family-Based Specification. A *family-based specification* defines behavior for a subset of product variants in the product line. When defining family-based specifications, a *presence condition* (propositional logic formula over the set of features) states for which combinations of features a specification should hold. With that, family-wide, product-based, and feature-based specifications can be generalized (i.e., expressed as a family-based specification) (Thüm et al., 2014a). For design-by-contract specifications, there is work that encodes feature-based specifications into family-based specifications to increase the proof efficiency of analyzing the product line (Thüm et al., 2012; Hähnle and Schaefer, 2012). In Section 5.2.1, we proposed variable predicates as a family-based specification technique. Since the goal of the proposed concepts from (Thüm et al., 2012; Hähnle and Schaefer, 2012) is to optimize verification effort and the encoding is not necessarily visible to the developer, the encoded family-based specifications are usually long and not easy to understand. However, our goal for variable predicates is to define a family-based specification upfront to gain more precise and readable specifications to start with.

Besides design-by-contract specifications, family-based specification techniques have been used for product lines defined by labelled transition systems (LTS) that model the behavior (Asirelli et al., 2012; Cordy et al., 2012b; Cordy et al., 2012a; Cordy et al., 2013b; Cordy et al., 2013a; Classen et al., 2013; Classen et al., 2014) and are analyzed with model checkers. These approaches use feature attributes in combination with temporal logic properties to analyze the product line behavior on the level of states in the LTS. The goal of variable predicates is to specify product lines on code level more precisely, but still in a readable way.

Also other areas, such as variability-aware datalog reasoning and product line transformation, have applied presence conditions to other kinds of specifications (e.g., transformations or facts) to show satisfiability properties (Shahin et al., 2023; Shahin and Chechik, 2020; Strüber et al., 2018).

Variable predicates, as proposed in this chapter, are a family-based specification technique because they are defined using presence conditions. Feature-based specification techniques are sufficient to specify the varying behavior of a method in one feature, but it is not possible to specify the behavior for a subgroup of products (i.e., a part of the family) when a methods' functionality depends on more than this one feature. This is what is usually done by using family-based specifications. However, to the best of our knowledge, there is no concept to define *readable and maintainable family-based specifications for design-by-contract specifications* yet. Variable predicates provide a lightweight and readable family-based specification technique that can be added to complement existing feature-based specification techniques of any realization technique, in our case the feature-based original predicates, to specify the behavior for a part of the family and therefore make specifications for software product lines more precise.

5.6.2. Software Product Line Verification

Refinement-based Software Construction Approaches for Software Product Lines. To the best of our knowledge, there is only one other work regarding the development of software product lines being phrased as CbC, which has been proposed by Pham (2017). The author generates correct-by-construction product variants from a product line by composing proof artifacts that have been created by a post-hoc verification framework. Hence, this idea of CbC is rather the assembly of post-hoc proof artifacts than extending the refinement rules of CbC itself, as we did. Thüm et al. (2011b) also propose an approach for proof composition for feature-oriented product lines written in Java. They use JML for specification and compose proof obligations for the proof assistant Coq (Coq, 1989/2025).

A related refinement-based software construction approach that we already discussed in Section 1.1 is Event-B, which has also been extended for the development of software product lines (Sorge et al., 2010; Gondal et al., 2011; Ait Wakrime et al., 2018). Generally, Event-B operates on system-level automata and CbC on code-level specifications, which differentiates these two approaches from one another. Event-B has been extended on the feature modeling level by mapping Event-B components to features (Sorge et al., 2010) and with rules for the composition of Event-B components accompanied by proof obligations that need to be checked when a concrete product variant is composed (Gondal et al., 2011). Rather than changing the feature modeling to map CbC implementations to features, we used the already existing notion of feature modules and investigated how to extend CbC and the set of refinement rules to achieve variability in specifications and implementations inside these feature modules. Since the abstraction levels of Event-B and CbC are so different, their concepts are too. The only commonality is the overall goal of realizing correct-by-construction software product line development.

Product-based Verification. With *product-based verification*, every product variant of a software product line is verified individually. This does not scale for large-scale software product lines. In Section 5.3.1, we proposed an optimized product-based verification strategy that might scale better, but the set of relevant feature sequences that our verification strategy checks is highly dependent on the structure of the software product line, meaning that in the worst case, still all product variants of a software product line might need to be checked. Bruns et al. (2011b) proposed an optimized product-based verification strategy for the software product line realization technique *delta-oriented programming* (Schaefer et al., 2010). Delta-oriented programming implements features in delta modules, which change the functionality, for example, by adding or removing fields or methods. Bruns et al. (2011b) propose *delta-oriented slicing*, where the idea is to determine the parts of the proof of a core module that have to be re-proven for a product variant to which deltas were applied.

Partial proofs allow proof reuse for product-based verification (Klebanov, 2006; Beckert and Klebanov, 2004; Kuiter et al., 2022). Therefore, they can also be categorized as an optimized product-based verification strategy. The idea is to split proofs at the method call level and then resolving these calls differently depending on the called method. Building on partial proofs, we propose *proof plans* in related work (Kuiter et al., 2022; Kuiter, 2020), which split verification tasks into smaller proofs. These smaller proofs can be reused across the verification of different product variants. Proof plans define a wide range of possible partial proofs, as proofs can be split into a large set of small proofs. The efficiency of an overall proof is determined by choosing a path in the proof plan that reuses as many proof parts as possible.

Proof repositories (Bubel et al., 2016) aim to optimize the verification of method calls and, thus, increase the proof efficiency of software product lines. A proof repository contains the bindings of method calls with a called implementation and indicates which bindings have been proven successfully. This information enables a high level of proof reuse. However, proof repositories have only been discussed theoretically for software product line verification as one use case. Building on the concepts of proof plans and proof repositories, we propose another approach for *partial proofs* that divides the verification in a software product line into a feature-specific proof start and a product-specific proof completion phase (Kodetzki et al., 2024). This concept of partial proofs has even been extended even further by splitting the proofs feature-wise (Flaschenträger, 2024).

In summary, we see potential in using partial proofs as an alternative optimized product-based verification strategy to our proposed optimized product-based verification strategy as proposed in this chapter. Since our refinement rules can be extended with additional strategies, we plan to incorporate partial proofs in future work and extend the evaluation to include a side-by-side evaluation of these verification strategies.

Family-based Verification. In *family-based verification* of software product lines, one superimposed metaproduct is generated and verified. This metaproduct is able to simulate every product variant. Thüm et al. (2012) propose a family-based verification strategy for deductive verification to prove the correctness of software product lines efficiently. Their strategy is developed for product lines written in feature-oriented programming for Java and specified using JML. In Section 5.3.2, we adapted their strategy to fit CbC and formally defined a procedure to transform our feature-based method refinements into a metaproduct. In their evaluation, Thüm et al. (2012) could save 85% of the verification time for the considered subject system in contrast to product-based verification. In our evaluation (see Section 5.5), we found the difference to be true for CbC, that our product-based verification strategy was superior in terms of proof efficiency compared to the family-based verification strategy. Von Rhein et al. (2016) develop a formal model of variability encoding based on Featherweight Java (Igarashi et al., 1999) and prove that variability encoding preserves the product variant behavior with respect to a core set of language mechanisms.

Variability encoding has also been proposed for model checking of software product lines. Post and Sinz (2008) propose an approach called *lifting* that integrates configuration information into a conventional model checking approach and Apel et al. (2011) develop a tool suite for feature-aware verification based on standard model checking. One major disadvantage of family-based verification strategies is that with every change in code or specification, the metaproduct has to be regenerated and, thus, also completely re-verified (Thüm et al., 2014a).

Feature-based Verification. *Feature-based verification* is a modular verification strategy for verifying the correctness of software product lines, where each feature is verified in isolation. This reduces the verification effort when a feature's functionality or specification changes. However, since features interact with each other, feature-based verification is usually insufficient. Knüppel et al. (2020a) combine feature-based and family-based verification in a verification strategy called FeFaLution. Their concept is to build partial proofs for every feature and reuse them to verify feature interactions in the complete system efficiently. Their evaluation results demonstrate the potential for reuse, but reveal challenges with overhead for smaller subject systems.

5.7. Chapter Summary

In this chapter, we investigated how CbC can be extended to support the development of software product lines using feature-oriented programming as a specific product line realization technique. Software product line engineering is increasingly applied in industry, including safety-critical domains, where behavioral correctness is essential. However, the high degree of variability in product lines is still a challenge for verification. To address this, we extended CbC in three aspects: 1) variability in specifications through original predicates (mimicking the behavior of `original` calls from feature-oriented programming) and variable predicates as a family-based specification mechanism to improve the precision of the specifications while maintaining readability, 2) new CbC refinement rules to include the two variation points on code-level from feature-oriented programming, `original` calls and variational method calls, and 3) a product-based and a family-based verification strategy to ensure correctness of the new refinement rules. We implemented these concepts in VarCorC, an extension of CorC, that additionally includes object-orientation (as introduced in Chapter 3) for software product line development. Our evaluation shows that our concepts are feasible: all specification techniques and refinement rules work in combination, and both verification strategies were able to verify the correctness of the subject systems. Additionally, we compare the specification effort and proof efficiency of using CbC in VarCorC with post-hoc verification in KeY and found that CbC indeed increases the specification effort compared to post-hoc verification, but it improves proof efficiency. Furthermore, we found that the product-based verification strategy is more efficient in terms of proof nodes than the family-based one. We identified possible optimizations for the family-based verification strategy, such as extracting duplicate refinement steps or using disjunctive specifications. In future work, we aim to improve the family-based verification strategy and integrate partial proofs (Kodetzki et al., 2024; Flaschenträger, 2024) as an alternative verification strategy that improves proof efficiency through proof reuse. Finally, we plan a holistic evaluation of proof efficiency, combining all proposed extensions and comparing our product-based and family-based verification strategies to partial proofs and post-hoc verification strategies, such as FeFaLution (Knüppel et al., 2020a).

Part III.

Conclusion and Outlook

6. Conclusion

In this thesis, we presented three contributions that scale CbC in three dimensions to be applicable in modern software engineering. In the first contribution, we proposed an extension of CbC with an object model and a development process that enables the combined use of CbC and other quality assurance techniques. In the second contribution, we lowered the entry barrier for developers without formal methods background by introducing a three-level development process with understandable feedback. In the third contribution, we extended CbC to support variable code and specification constructs that enable the development of highly configurable software systems. In this chapter, we conclude this thesis by answering our three sub-research questions as well as our main research question. Furthermore, we discuss potential future work.

6.1. Contribution

In the following, we answer our initially defined research questions based on the concrete contributions that we presented in this thesis.

RQ1: How can we scale Correctness-by-Construction to object-oriented programs, and how can we incorporate Correctness-by-Construction into an integrated software development process?

With this research question, we addressed two challenges: First, CbC was initially developed solely to create isolated algorithms, which can only be applied to a limited extent in modern software engineering, where more complex code structures, such as object-orientation, are used. Second, CbC could hardly be combined with existing software development processes, hindering its adoptability in practice. With CorC 2.0, we extended CorC's programming model by objects as used in object-oriented programming to tackle the first challenge. We extended CorC by three features that allow the creation of object-oriented Java programs using CbC. First, we implemented a graphical editor to create classes with fields, class invariants, and methods. Second, we support the constructive development of interfaces and incorporated inheritance using the Liskov principle. Third, we included a framing condition for methods that preserve the modularity of the contracts and avoid unwanted side-effects of method calls. To tackle the second challenge, the integrability of CbC into existing software development processes, we implemented a roundtrip engineering process such that existing JAVA classes can be imported into CORC where their correctness can be shown, and exported back into the original JAVA project as verified JAVA code. Thereby, a developer can freely decide which parts are constructed using CbC and which parts are implemented and tested/verified with other methods (e.g., depending on the safety criticality of that part). CORC 2.0 was evaluated by implementing three subject systems. We compared the proof efficiency in terms of proof nodes and verification time using CbC as implemented in CORC 2.0 with post-hoc verification as implemented in KEY. The evaluation results do not indicate that one of the approaches, CbC or post-hoc verification, is superior in terms of proof efficiency and usability. Instead, we conclude that their combined use can leverage the individual benefits of both approaches. In fact, this underlines our hypothesis that the combined use of different quality assurance techniques is a good strategy to develop correct software. The introduction of objects to CbC and the implementation of a roundtrip engineering process were crucial to achieving this goal for CbC and CORC 2.0.

RQ2: How can we lower the entry barrier for non-experts in by-Construction development?

Software testing is state-of-the-art for ensuring the quality of a software system, although formal methods, such as CbC, provide stronger guarantees of functional correctness. By introducing three levels of correctness guarantees that include an intermediate level *tested*, we lowered the entry barrier to using CbC for non-experts in software verification, with the goal of broadening the user group of CbC. The first level, *specified*, is supported by new error handling to detect errors in syntax. The second level, *tested*, is new for CbC-based development and is supported by test case generation from the specification of the first level. Test cases show major errors in a program and provide more readable feedback to the developer. The third level, *verified*, is supported by counterexample generation. Counterexamples help to detect errors more easily than examining open proof goals. We evaluated our three-level process as implemented in CORC quantitatively and qualitatively. In the quantitative evaluation, we found that almost three-quarters of defects introduced by mutating the source code could already be located at the level *tested*, which is presumably easier to understand for non-experts in software verification. In the qualitative evaluation, we performed a usability refinement study, which resulted in a list of helpful suggestions for improvement. Furthermore, the participants rated the extensions at each correctness level as useful. In conclusion, we have shown that successive levels of correctness guarantees do indeed support the development of verified programs using CbC for non-experts in software verification.

RQ3: How can we extend Correctness-by-Construction to support the development of highly configurable software systems?

Software product line engineering provides systematic reuse and variability mechanisms to develop whole program families efficiently. CbC in its original form does not support these specific reuse and variability mechanisms. To address this, we extended CbC in three aspects: 1) Variability in specifications through original predicates (mimicking the behavior of the reuse mechanism for feature-oriented programming, the `original` call, on code level) and variable predicates as a family-based specification mechanism. These improve specification precision while maintaining readability. 2) New CbC refinement rules to include the two variation points on code level: `original` calls and variational method calls. 3) A product-based and a family-based verification strategy to ensure correctness of the new refinement rules. We implemented these concepts in VarCorC, an extension of CorC, that additionally includes object-orientation (as introduced in Chapter 3) for program family development. Our evaluation showed that our concepts are feasible: all specification mechanisms and refinement rules work in combination, and both verification strategies allowed us to verify the correctness of the subject systems. Additionally, we compared the specification effort and proof efficiency of using CbC in VarCorC with post-hoc verification in KeY and found that CbC indeed improves the proof efficiency compared to post-hoc verification, but it also increases the specification effort. Furthermore, we found that the product-based verification strategy is more efficient in terms of proof nodes than the family-based verification strategy. We identified potential optimizations for the family-based verification strategy, such as extracting duplicate refinement steps or using disjunctive specifications. In conclusion, we have shown that by extending CbC with our concepts for variability in the specifications, new refinement rules, and verification strategies for showing the correctness of the new refinement rules, we are able to develop correct-by-construction program families using the implementation in VarCorC.

Main Research Question

How can we scale Correctness-by-Construction from developing isolated algorithms to be applicable in modern software engineering?

Modern software engineering poses both functional and non-functional challenges to the practical adoption of CbC, such as missing support for certain programming paradigms and challenges regarding usability and process integration. We identified four specific challenges for CbC in modern software engineering and defined three research questions to address these four challenges by scaling CbC in three dimensions. We were able to answer all three of our initially defined research questions by proposing and evaluating three contributions tailored to the challenges that the research questions are based on. We addressed the first challenge of missing scalability to larger software systems than isolated algorithms by introducing a programming model with objects and the second challenge of a missing integrated software development process by incorporating CbC into a three-step roundtrip engineering process. Both concepts are implemented in the tool CorC 2.0 building the first contribution, which scales CbC's *applicability* to the development of object-oriented systems while also enhancing the *adoptability* of CbC program development by enabling collaborative use with other quality assurance techniques. We addressed the third challenge that CbC has a high entry barrier for non-experts in software verification, as experts in software verification are rare in the field, by incorporating testing as an intermediate level between specifying and verifying CbC programs. With that, we improved the *accessibility* of CbC for non-experts in software verification. We addressed the fourth challenge of missing support for advanced programming paradigms, such as feature-oriented programming for highly configurable software, by incorporating variable constructs in three core aspects: the specification, the refinement rules, and verification strategies to show the correctness of the whole system for every program variant.

In conclusion, we answer our main research question by combining all three contributions in the tool CorC. This enables the development of object-oriented systems and software product lines within software teams using existing software development processes and varying expertise levels.

6.2. Future Work

Modern software engineering evolves rapidly, with new programming paradigms and languages constantly emerging. Consequently, it is important to also adopt formal method technologies and tools to the new challenges of upcoming techniques. Our vision for CorC is to become an ecosystem that is able to tackle many of these future challenges while leveraging the benefits of by-Construction engineering. Some of the most promising directions of future work are discussed in the following.

Guaranteeing Event Trace Properties using Correctness-by-Construction. Since safety- and security-critical systems generally have a higher demand for behavioral correctness, this is a typical application domain for CbC. An example of a specific type of safety and security requirement is protocol properties, such as *event trace properties*. Event trace properties usually specify that certain sequences of events are not allowed to happen in a system. For example, in an elevator, it might be specified that the door is never opened before a floor has been reached. On system level, event trace properties are often proven by model checking (Sistla et al., 2000) or protocol verification (Mota et al., 2023). On code level, where CbC is applied, histories are modeled to guarantee that certain event traces do not occur (Hiep et al., 2020b; Hiep et al., 2020a). However, it is a time-consuming and challenging process to model method invocations inside an object as an event history list and then to verify the absence of event traces on this list. This can be circumvented by specifying event trace properties in advance and then using CbC to refine the program out of that specification, guaranteeing that the sequence of events has not happened. A promising future research direction is to investigate how the specification language of CbC and its refinement rules can be extended such that event trace properties can be specified and verified in a light-weight way to be able to guarantee event trace properties by-Construction that can be used to model specific security and safety properties.

Feature interactions in software product lines are another valuable application area for event sequence properties. Naturally, features in a software product line interact with each other by using shared variables or by calling methods defined in other features. While most of these feature interactions are wanted or even required 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 (Hall, 2005) is when a feature that automatically forwards incoming emails to a certain address and a feature that decrypts incoming emails 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 emails. While we can already guarantee that correct-by-construction software product lines, as proposed in Chapter 5, are free of functionally unwanted feature interactions based on shared variables, we found that most feature interaction problems, like the one described in the email example, cannot be specified and verified with our state-based specification language, as sequences of method calls cause the unwanted interactions. Hence, event sequences are a promising solution for guaranteeing that no unwanted feature interactions occur.

Beyond Functional Correctness: X-by-Construction. In this thesis, we only discussed CbC for guaranteeing the functional correctness of software systems. However, software quality also requires *non-functional quality properties* beyond functional correctness, such as performance, reliability, security, and maintainability. Ter Beek et al. (2018) defined the term *X-by-Construction*, where X stands for any non-functional property. In the previous paragraph, we already mentioned event sequence properties, which can be categorized as safety or security properties. Furthermore, CbC has been applied to information flow properties (Runge et al., 2022) that are also used to specify security properties. Besides security, other non-functional properties of software quality, such as performance and reliability, are still unsupported. For future research, integrating concepts to guarantee other non-functional properties alongside functional correctness is a promising direction. Integrating further non-functional properties would bring X-by-Construction to the next level, where different non-functional properties (X's) can be selected according to the requirements of a software project and developed in one by-Construction approach, guaranteeing both the required non-functional properties and functional correctness. In this way, the development of high-quality software through refinement-based construction with provable trade-offs between functional and non-functional properties is leveraged.

Concurrent Programs by-Construction. Another popular programming paradigm that is used in modern software engineering is *concurrency*. Some advantages of concurrent programs are faster processing and improved memory utilization. In practice, even small applications contain concurrency, e.g., in communication with web services or databases, but concurrency is also an essential part of embedded systems. Besides the aforementioned advantages of concurrency, it also yields many new challenges in terms of reliability and data consistency. In CbC, we currently do not support the construction of concurrent programs. For this, a formalized concurrency model, such as ABS (Johnsen et al., 2011) is needed, and additional CbC refinement rules (e.g., for asynchronous method calls) need to be defined. Not only functional correctness, but also ensuring confidentiality and integrity of data is an important concern where we see potential to leverage the benefits of a refinement-based software development approach, like CbC.

User Studies in Industry and Case Studies for the Evaluation of CbC. One overarching goal of our research on CbC is to bridge the gap between theoretical approaches, such as CbC, and their application in practice. To assess whether CbC is a real alternative to post-hoc verification in terms of quantitative metrics, like proof efficiency, and qualitative metrics, like usability, we need to further evaluate CbC and our tool CorC. For this, it is crucial to design larger subject systems such that the scalability of proposed concepts can be assessed. Furthermore, qualitative user studies with experts in industry that run over a longer period of time would bring detailed insights into the obstacles of applying CbC in real-world processes and generate improvements in the tool from a practical point of view. With our extension of CorC, we created a basis for further evaluation that may hopefully answer some of these questions.

Language and Tool Versatility. Even though the theoretical approach of CbC is not tailored to a specific programming language, our tool CorC only supports the development of Java programs. In practice, software projects are usually implemented with multiple programming languages, and especially in safety-critical systems, other programming languages, such as C/C++ and Rust, are used. Therefore, it would be beneficial to extend CorC to support additional programming languages. To include CbC development for other programming languages in CorC, the editor must support the syntax of the programming language and its corresponding specification language. To guarantee functional correctness, the side conditions of the refinement rules must be translated into proof obligations checkable by a program verifier for the new programming language. The current format of the generated proof obligations is tailored to KeY and must be adopted. This would significantly broaden the practical applicability of CbC, especially in domains where languages like C and Rust dominate.

Part IV.

Appendix

A. Detailed Evaluation Results for Chapter 3

In the following, we present the detailed evaluation results broken down into the verification time (average over five runs) and proof nodes per method of the *BankAccount*, *Email*, and *Elevator* subject systems.

Method	#Proof Nodes CbC	#Proof Nodes PhV	Verification Time in ms CbC	Verification Time in ms PhV
undoUpdate	205	183	730,75	747,3
update	181	188	626,75	627,67
creditAccount	25	45	104,5	132,67
dailyUpdate	467	636	1990,5	2261
interestCalculate	422	Unclosed	1164	Unclosed
transactionLock	432	663	2253	1995,67
transactionTransfer	799	989	3556,75	3323,3
unLock	21	41	104,5	175,67
interestNextDay	233	381	1066,25	1154,67
interestNextYear	191	239	836,75	786,3
constructClient	50	68	207,75	292,3
createClient	1391	Unclosed	7108,5	Unclosed
getClientByAdress	1420	Unclosed	4509,75	Unclosed
getClientById	67	Unclosed	210,75	Unclosed
outgoing	67	61	212,75	200
resetClients	673	Unclosed	2450,75	Unclosed
constructEmail	36	42	111,25	156,33
createEmail	558	772	2591	2996,67
setEmailBody	38	47	110,25	185
setEmailFrom	40	51	104,5	192,33
setEmailSubject	38	47	113,25	170,67
setEmailTo	38	47	111,5	173,67
areDoorsOpen	64	41	229,25	141,67
buttonIsPressed	70	69	206	144
enterElevatorBase	805	1092	4829,5	4577
enterElevator	457	2315	2370	8710
pressButtonBase	87	94	317,25	529
pressButton	97	109	444,25	503,33
resetFloorButton	79	84	336,25	287
reverse	192	108	985,25	338,33
stopRequestedBase	878	1150	3179	4367,67
createEnvironment	1367	Unclosed	4375,5	Unclosed
isTopFloor	86	97	255,5	345,33
addWaitingPerson	4768	1041	16012,25	4320,33
callElevator	28	42	105,5	174,67
createFloor	346	1066	1641,75	4027
reset	29	42	106,25	173,67
createPerson	3835	1764	12323	6171,67
PersonenterElevator	163	134	432	393,67
PersonleaveElevator	30	44	105,25	174,67

Table A.1.: Detailed evaluation results of Chapter 3 (Bordis et al., 2022a). Verification time and proof nodes of all methods from *BankAccount*, *Email*, and *Elevator* subject systems. Verification time is averaged over five runs.

B. Detailed Evaluation Results for Chapter 4

B.0.1. Mutation Operations in CorC

CorC supports the following mutation operations:

- AORB: a + b $\rightarrow$ M_1: a - b M_2: a * b M_3: a / b
- AORS: i++ $\rightarrow$ M_1: i– M_2: ++i M_3: –i
- AOIU: a < b $\rightarrow$ M_1: -a < b M_2: a < -b ...
- ROR: a < b $\rightarrow$ M_1: a <= b M_2: a >= b ...
- COR: a || b $\rightarrow$ M_1: a && b M_2: a & b ...
- COD: !a $\rightarrow$ M_1: a
- SOR: a > > b $\rightarrow$ M_1: a < < b M_2: a > > > > b
- LOR: a | b $\rightarrow$ M_1: a & b M_2: aˆb
- LOD: ˜a $\rightarrow$ M_1: a
- ASRS: a += b $\rightarrow$ M_1: a -= b M_2: a *= b ...
- Pre Boundaries: ==, >, <, >=, <=, &&, forall
- Post Boundaries: >=, <=, !=, >, <, ||, exists

B.0.2. Interview Protocols

We provide the protocols in Tables B.1 to B.5. The answers have been documented in the form of bullet points during the interview.

Participant 1 (Group A)	
General Questions before Interview	*Answers*
To what extent have you used CorC and KeY before?	Used KeY and CorC in their bachelor's thesis Roughly 3 – 4 months experience
Are you familiar with the algorithms in the tasks?	Thinks they have achieved a decent proficiency Feels comfortable using CorC Feels a little less comfortable using KeY
Do you have experience with software testing?	Has seen all problems LinearSearch and MaxElement are most familiar, the other ones less
Do you have experience with counterexample generation?	Never used or heard of before
Questions After Experiment for Level "Specified"	
Were the error messages useful to locate the defect?	The “Rust-like” tips helped a lot Found the explanation of the error quite useful, although it did not help them a lot
Did the error messages speed up the debugging?	Yes
How can we improve error messages? Are there any features missing?	None
Would you use error messages instead of the original version of CorC?	Yes
Questions After Experiment for Level "Tested"	
Was the test case generation useful to locate the defect?	Yes, because the test cases provide a different view on the error. It gives more information to solve the problem. The test cases provide a quick overview, if the method is working Paths show up to which point in the program is working
Did the test case generation speed up the debugging?	Yes, less guessing
How can we improve test case generation? Are there any features missing?	A graphical representation of the tested refinements
Would you use test case generation instead of the original version of CorC?	Yes
Questions After Experiment for Level "Verified"	
Was the counterexample generation useful to locate the defect?	Pretty useful once you get used to them Syntax is horrible to read
Did the counterexample generation speed up the debugging?	First not, but after explanation probably
How can we improve counterexample generation? Are there any features missing?	Improving the syntax, the numbers in the variable names are confusing
Would you use counterexample generation instead of the original version of CorC?	It offers more information; you may not need to go into the KeY files
General Questions after Last Experiment	
Did the defects in the tasks seem realistic to you?	The errors are realistic
Did you see any of these defects before when using CorC?	These careless mistakes do happen from time to time (especially things like forgetting a statement)

Table B.1.: Answers in the interview of Participant 1.

Participant 2 (Group B)	
General Questions before Interview	*Answers*
To what extent have you used CorC and KeY before?	Experienced Used CorC for bachelor's thesis, project work and master's thesis Also familiar with KeY
Are you familiar with the algorithms in the tasks?	Very familiar with LinearSearch and MaxElement Familiar with the others
Do you have experience with software testing?	Yes, a bit
Do you have experience with counterexample generation?	Never used before
Questions After Experiment for Level "Specified"	
Were the error messages useful to locate the defect?	Especially for new users Already has a feeling where exceptions come from
Did the error messages speed up the debugging?	Yes, more information in console available
How can we improve error messages? Are there any features missing?	None
Would you use error messages instead of the original version of CorC?	Yes
Questions After Experiment for Level "Tested"	
Was the test case generation useful to locate the defect?	Yes, especially with complex programs to check whether the program can in any way be correct
Did the test case generation speed up the debugging?	Yes, for complex programs
How can we improve test case generation? Are there any features missing?	Graphical representation of tested statements Marking the parts of the postcondition that are used in assertions
Would you use test case generation instead of the original version of CorC?	Yes
Questions After Experiment for Level "Verified"	
Was the counterexample generation useful to locate the defect?	Not used counterexamples, may need more time to be comfortable and use it to its fullest extent Got used to their own approach of fixing faults in CorC, thus this extension might be less useful
Did the counterexample generation speed up the debugging?	Syntax is hard-to-read Yes, if you get used to it / know how to use it
How can we improve counterexample generation? Are there any features missing?	Better syntax for counterexample generation Filter more unnecessary / dummy variables Button for proving without counterexample generation for faster execution time
Would you use counterexample generation instead of the original version of CorC?	Yes, but has to get used to it
General Questions after Last Experiment	
Did the defects in the tasks seem realistic to you?	Yes, getting variant wrong or similar mistakes can happen, especially when working with loops Forgetting a statement (like in the logarithm problem) more unlikely
Did you see any of these defects before when using CorC?	Yes

Table B.2.: Answers in the interview of Participant 2.

Participant 3 (Group A)	
General Questions before Interview	*Answers*
To what extent have you used CorC and KeY before?	Has been working with CorC and KeY for a few years Used the programs in university
Are you familiar with the algorithms in the tasks?	Very familiar with MaxElement and LinearSearch Has already seen Logarithm and Factorial, but is less familiar with them
Do you have experience with software testing?	Yes
Do you have experience with counterexample generation?	Never used
Questions After Experiment for Level "Specified"	
Were the error messages useful to locate the defect?	Is useful, as you have list of potential error sources that you can go through, especially if you are less experienced with CorC
Did the error messages speed up the debugging?	Could be time saving because no need to open KeY
How can we improve error messages? Are there any features missing?	Lacks some information compared to KeY (e.g., location of the problematic file)
Would you use error messages instead of the original version of CorC?	Yes
Questions After Experiment for Level "Tested"	
Was the test case generation useful to locate the defect?	Yes, because you can see what happens when executing the refinements Test cases are easy to understand
Did the test case generation speed up the debugging?	Yes
How can we improve test case generation? Are there any features missing?	Graphical representation of tested refinements Highlighting the tested statement (Path not enough) Highlighting the assertions Separating output of test case generation from CorC's output
Would you use test case generation instead of the original version of CorC?	Yes, because locating bugs with KeY has sometimes been difficult Especially when there is more than one bug
Questions After Experiment for Level "Verified"	
Was the counterexample generation useful to locate the defect?	Analyzing the pre/postcondition with a counter example clears things up Helps to pinpoint the exact problem
Did the counterexample generation speed up the debugging?	Yes
How can we improve counterexample generation? Are there any features missing?	Clean counterexample output, improve syntax Turn the counterexample generation on / off
Would you use counterexample generation instead of the original version of CorC?	Yes, only adds information
General Questions after Last Experiment	
Did the defects in the tasks seem realistic to you?	Yes ProofInputException happens often Errors in the implementation / specification too
Did you see any of these defects before when using CorC?	Yes

Table B.3.: Answers in the interview of Participant 3.

Participant 4 (Group B)	
General Questions before Interview	*Answers*
To what extent have you used CorC and KeY before?	Wrote bachelor's thesis about CorC Never really developed something with CorC Not really familiar with KeY's logic
Are you familiar with the algorithms in the tasks?	LinearSearch and MaxElement are familiar The rest not
Do you have experience with software testing?	Yes
Do you have experience with counterexample generation?	No
Questions After Experiment for Level "Specified"	
Were the error messages useful to locate the defect?	Yes, the bullet points were helpful Otherwise would not have a clue where to look
Did the error messages speed up the debugging?	Definitely
How can we improve error messages? Are there any features missing?	No
Would you use error messages instead of the original version of CorC?	Yes
Questions After Experiment for Level "Tested"	
Was the test case generation useful to locate the defect?	Yes, but understanding the path was difficult first
Did the test case generation speed up the debugging?	Yes
How can we improve test case generation? Are there any features missing?	Different representation of the path Graphical representation that a statement has been tested
Would you use test case generation instead of the original version of CorC?	Yes, when something does not verify
Questions After Experiment for Level "Verified"	
Was the counterexample generation useful to locate the defect?	Having a concrete example which does not work makes it way simpler to spot the error
Did the counterexample generation speed up the debugging?	Yes
How can we improve counterexample generation? Are there any features missing?	Improve the counterexample generation output Maybe print the values given by the counter example into the pre- and postcondition for faster identification of error in the postcondition
Would you use counterexample generation instead of the original version of CorC?	Yes, more information is always better
General Questions after Last Experiment	
Did the defects in the tasks seem realistic to you?	Generally yes The Logarithm task was too obvious
Did you see any of these defects before when using CorC?	No (not really developed with CorC)

Table B.4.: Answers in the interview of Participant 4.

Participant 5 (Group B)	
General Questions before Interview	*Answers*
To what extent have you used CorC and KeY before?	Has used CorC and KeY a lot More familiar with CorC than KeY
Are you familiar with the algorithms in the tasks?	Yes
Do you have experience with software testing?	Yes
Do you have experience with counterexample generation?	Never used
Questions After Experiment for Level "Specified"	
Were the error messages useful to locate the defect?	Would probably help
Did the error messages speed up the debugging?	Already knows where to look when an exception is thrown
How can we improve error messages? Are there any features missing?	No
Would you use error messages instead of the original version of CorC?	Yes, no drawbacks
Questions After Experiment for Level "Tested"	
Was the test case generation useful to locate the defect?	Yes, after getting used to the output Separating complex postconditions into smaller parts is very useful
Did the test case generation speed up the debugging?	Yes
How can we improve test case generation? Are there any features missing?	An additional Icon for failed test Visualizing which Refinement was translated to which part of code Visualizing the path Grouping of failed and passed test cases
Would you use test case generation instead of the original version of CorC?	Yes
Questions After Experiment for Level "Verified"	
Was the counterexample generation useful to locate the defect?	Needed more time to understand counterexample (compared to already familiar ways to locate the error) Output needs to be improved
Did the counterexample generation speed up the debugging?	For me not Probably when output has been improved
How can we improve counterexample generation? Are there any features missing?	Improving the output (syntax and filtering)
Would you use counterexample generation instead of the original version of CorC?	Yes No drawbacks as nothing got removed
General Questions after Last Experiment	
Did the defects in the tasks seem realistic to you?	Yes
Did you see any of these defects before when using CorC?	Yes

Table B.5.: Answers in the interview of Participant 5.

C. Detailed Evaluation Results for Chapter 5

In the following, we provide the data that the evaluation in Chapter 5 is based on. In Table C.1, we present the total number of conjuncts in the specification conditions to answer research question **RQ2**. In Table C.2, we present the total number of proof nodes for CbC and post-hoc verification for research question **RQ3**. In Table C.3, we present the total number of proof nodes and the total number of CbC diagram nodes for the product-based verification strategy and the family-based verification strategy for research question **RQ4**.

Methods	Correctness-by-Construction				JML Specifications (Post-hoc Verification)		
	Preconditions	Postconditions	Intermediate Conditions	Global Conditions	Preconditions	Postconditions	Class Invariants
push	3	8	11	12	3	8	3
update	2	4	11	8	2	4	2
undoUpdate	2	4	11	8	2	4	2
nextDay	3	4	2	4	3	4	1
nextYear	2	3	1	2	2	3	1

Table C.1.: Detailed evaluation results of research question **RQ2** in Chapter 5. Total number of conjuncts in specification conditions split up into preconditions, postconditions, global conditions, and intermediate conditions for CbC and in preconditions, postconditions, and class invariants for JML specifications (post-hoc verification).

Method	#Proof Nodes CbC	#Proof Nodes PhV
push (6 variants)	12050	5280
update (2 variants)	2615	1228
undoUpdate (2 variants)	2600	1034
nextDay (4 variants)	3250	592
nextYear (2 variants)	844	208

Table C.2.: Detailed evaluation results of research question **RQ3** in Chapter 5. Total number of proof nodes of all methods containing at least one `original` call from *IntegerList* and *BankAccount* subject systems.

Methods	Number of Proof Nodes		CbC Diagram Nodes	
	Product-based	Family-based	Product-based	Family-based
push (6 variants)	4293	9402	17	54
sort (2 variants)	8406	11878	28	30
update (2 variants)	863	1138	24	32
undoUpdate (2 variants)	964	1175	24	32
nextDay (4 variants)	324	346	10	22
nextYear (2 variants)	102	131	6	10

Table C.3.: Detailed evaluation results of research question **RQ4** in Chapter 5. Total number of proof nodes and CbC diagram nodes for all methods from *IntegerList* and *BankAccount* subject systems.

References

Abrial, Jean-Raymond (2010). *Modeling in Event-B: System and Software Engineering.* Cambridge ; New York: Cambridge University Press.

Abrial, Jean-Raymond, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin (Nov. 2010). “Rodin: An Open Toolset for Modelling and Reasoning in Event-B”. In: *International Journal on Software Tools for Technology Transfer* 12.6, pp. 447–466. DOI: 10.1007/s10009-010-0145-y.

Agostinho, Sérgio, Ana Moreira, and Pedro Guerreiro (Mar. 2008). “Contracts for Aspect-Oriented Design”. In: *Proceedings of the 2008 AOSD Workshop on Software Engineering Properties of Languages and Aspect Technologies*. Brussels Belgium: ACM, pp. 1–6. DOI: 10.1145/1408647.1408648.

Ahrendt, Wolfgang, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich (2014). “The KeY Platform for Verification and Analysis of Java Programs”. In: *Verified Software: Theories, Tools and Experiments*. Vol. 8471. Cham: Springer International Publishing, pp. 55–71. DOI: 10.1007/978-3-319-12154-3_4.

Ahrendt, Wolfgang, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (2016a). *Deductive Software Verification – The KeY Book*. Vol. 10001. Lecture Notes in Computer Science. Cham: Springer. DOI: 10.1007/978-3-319-49812-6.

Ahrendt, Wolfgang, Christoph Gladisch, and Mihai Herda (2016b). “Proof-Based Test Case Generation”. In: *Deductive Software Verification – The KeY Book*. Vol. 10001. Cham: Springer International Publishing, pp. 415–451. DOI: 10.1007/978-3-319-49812-6_12.

Ahrendt, Wolfgang, Dilian Gurov, Moa Johansson, and Philipp Rümmer (2022). “TriCo—Triple Co-piloting of Implementation, Specification and Tests”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*. Vol. 13701. Cham: Springer International Publishing, pp. 174–187. DOI: 10.1007/978-3-031-19849-6_11.

Ait Wakrime, Abderrahim, J Paul Gibson, and Jean-Luc Raffy (June 2018). “Formalising the Requirements of an E-Voting Software Product Line Using Event-B”. In: *2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. Paris: IEEE, pp. 78–84. DOI: 10.1109/WETICE.2018.00022.

Ambert, F., F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and Mark Utting (2002). “BZ-TT: A Tool-set for Test Generations from Z and B Using Constraint Logic Programming”. In: *Proceedings of the CONCUR ’02 Workshop on Formal Approaches to Testing of Software (FATES ’02)*. Brno, Czech Republic, pp. 105–120.

America, Pierre (1991). “Designing an Object-Oriented Programming Language with Behavioural Subtyping”. In: *Foundations of Object-Oriented Languages*. Vol. 489. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 60–90. DOI: 10.1007/BFb0019440.

Ammann, P., J. Offutt, and Hong Huang (2003). “Coverage Criteria for Logical Expressions”. In: *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* Denver, Colorado, USA: IEEE, pp. 99–107. DOI: 10.1109/ISSRE.2003.1251034.

Ammann, Paul and Jeff Offutt (2017). *Introduction to Software Testing*. Edition 2. Cambridge, United Kingdom ; New York, NY, USA: Cambridge University Press.

Andrews, J.H., L.C. Briand, Y. Labiche, and A.S. Namin (Aug. 2006). “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria”. In: *IEEE Transactions on Software Engineering* 32.8, pp. 608–624. DOI: 10.1109/TSE.2006.83.

Apel, Sven, Don Batory, Christian Kästner, and Gunter Saake (2013a). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, Heidelberg: Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-37521-7.

Apel, Sven, Christian Kastner, and Christian Lengauer (Jan. 2013b). “Language-Independent and Automated Software Composition: The FeatureHouse Experience”. In: *IEEE Transactions on Software Engineering* 39.1, pp. 63–79. DOI: 10.1109/TSE.2011.120.

Apel, Sven and Christian Lengauer (2008). “Superimposition: A Language-Independent Approach to Software Composition”. In: *Software Composition*. Vol. 4954. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 20–35. DOI: 10.1007/978-3-540-78789-1_2.

Apel, Sven, Hendrik Speidel, Philipp Wendler, Alexander Von Rhein, and Dirk Beyer (Nov. 2011). “Detection of Feature Interactions Using Feature-Aware Verification”. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Lawrence, KS, USA: IEEE, pp. 372–375. DOI: 10.1109/ASE.2011.6100075.

Apel, Sven, Alexander Von Rhein, Thomas Thüm, and Christian Kästner (Aug. 2013c). “Feature-Interaction Detection Based on Feature-Based Specifications”. In: *Computer Networks* 57.12, pp. 2399–2409. DOI: 10.1016/j.comnet.2013.02.025.

Asirelli, Patrizia, Maurice H. Ter Beek, Alessandro Fantechi, and Stefania Gnesi (2012). “A Compositional Framework to Derive Product Line Behavioural Descriptions”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Vol. 7609. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 146–161. DOI: 10.1007/978-3-642-34026-0_12.

Back, Ralph-Johan (1998). *Refinement Calculus: A Systematic Introduction*. 1st ed. Texts in Computer Science Ser. New York, NY: Springer New York.

– (May 2009). “Invariant Based Programming: Basic Approach and Teaching Experiences”. In: *Formal Aspects of Computing* 21.3, pp. 227–244. DOI: 10.1007/s00165-008-0070-y.

Back, Ralph-Johan, Johannes Eriksson, and Magnus Myreen (2007). “Testing and Verifying Invariant Based Programs in the SOCOS Environment”. In: *Tests and Proofs*. Vol. 4454. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 61–78. DOI: 10.1007/978-3-540-73770-4_4.

Barnett, Mike, K. Rustan M. Leino, and Wolfram Schulte (2005). “The Spec# Programming System: An Overview”. In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Vol. 3362. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 49–69. DOI: 10.1007/978-3-540-30569-9_3.

Barrett, Clark, Aaron Stump, and Cesare Tinelli (2010). “The SMT-lib Standard: Version 2.0”. In: *Proceedings of the 8thInternational Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Vol. 13. Edinburgh, UK, p. 14.

Barrett, Clark and Cesare Tinelli (2018). “Satisfiability Modulo Theories”. In: *Handbook of Model Checking*. Cham: Springer International Publishing, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11.

Batory, D., J.N. Sarvela, and A. Rauschmayer (June 2004). “Scaling Step-Wise Refinement”. In: *IEEE Transactions on Software Engineering* 30.6, pp. 355–371. DOI: 10.1109/TSE.2004.23.

Batory, Don (2005). “Feature Models, Grammars, and Propositional Formulas”. In: *Software Product Lines*. Vol. 3714. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 7–20. DOI: 10.1007/11554844_3.

Batory, Don, Rich Cardone, and Yannis Smaragdakis (2000). “Object-Oriented Frameworks and Product Lines”. In: *Software Product Lines*. Boston, MA: Springer US, pp. 227–247. DOI: 10.1007/978-1-4615-4339-8_13.

Beck, Kent (2015). *Test-Driven Development: By Example*. 20. printing. The Addison-Wesley Signature Series. Boston: Addison-Wesley.

Beckert, B. and V. Klebanov (2004). “Proof Reuse for Deductive Program Verification”. In: *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004*. Beijing, China: IEEE, pp. 77–86. DOI: 10.1109/SEFM.2004.1347505.

Beckert, Bernhard, Vladimir Klebanov, and Benjamin Weiß (2016). “Dynamic Logic for Java”. In: *Deductive Software Verification – The KeY Book*. Vol. 10001. Cham: Springer International Publishing, pp. 49–106. DOI: 10.1007/978-3-319-49812-6_3.

Béust, Cedric (n.d.). *TestNG Documentation. https://testng.org/ Accessed 06 December 2024*. Documentation.

Beyer, Dirk and M. Erkan Keremoglu (2011). “CPAchecker: A Tool for Configurable Software Verification”. In: *Computer Aided Verification. (CAV 2011)*. Vol. 6806. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16.

Birkemeyer, Lukas, Tobias Pett, Andreas Vogelsang, Christoph Seidl, and Ina Schaefer (Feb. 2022). “Feature-Interaction Sampling for Scenario-based Testing of Advanced Driver Assistance Systems*”. In: *Proceedings of the 16th International Working*

Conference on Variability Modelling of Software-Intensive Systems. Florence Italy: ACM, pp. 1–10. DOI: 10.1145/3510466.3510474.

Blom, Stefan and Marieke Huisman (2014). “The VerCors Tool for Verification of Concurrent Programs”. In: *FM 2014: Formal Methods*. Vol. 8442. Cham: Springer International Publishing, pp. 127–131. DOI: 10.1007/978-3-319-06410-9_9.

Borgida, A., J. Mylopoulos, and R. Reiter (Oct. 1995). “On the Frame Problem in Procedure Specifications”. In: *IEEE Transactions on Software Engineering* 21.10, pp. 785–798. DOI: 10.1109/32.469460.

Bouquet, Fabrice, Frédéric Dadeau, Bruno Legeard, and Mark Utting (2005). “JML-Testing-Tools: A Symbolic Animator for JML Specifications Using CLP”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3440. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 551–556. DOI: 10.1007/978-3-540-31980-1_37.

Bourdonov, Igor B., Alexander S. Kossatchev, Victor V. Kuliamin, and Alexander K. Petrenko (2002). “UniTesK Test Suite Architecture”. In: *FME 2002:Formal Methods—Getting IT Right*. Vol. 2391. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 77–88. DOI: 10.1007/3-540-45614-7_5.

Bove, Ana, Peter Dybjer, and Ulf Norell (2009). “A Brief Overview of Agda – A Functional Language with Dependent Types”. In: *Theorem Proving in Higher Order Logics*. Vol. 5674. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6.

Brambilla, Marco, Manuel Wimmer, and Jordi Cabot (2017). *Model-Driven Software Engineering in Practice*. Second edition (Online-Ausg.) Synthesis Lectures on Software Engineering # 4. San Rafael, California: Morgan & Claypool. DOI: 10.1007/978-3-031-02549-5.

Briot, Jean-Pierre, Rachid Guerraoui, and Klaus-Peter Lohr (Sept. 1998). “Concurrency and Distribution in Object-Oriented Programming”. In: *ACM Computing Surveys* 30.3, pp. 291–329. DOI: 10.1145/292469.292470.

Brucker, Achim D. and Burkhart Wolff (2009). “HOL-TestGen: An Interactive Test-Case Generation Framework”. In: *Fundamental Approaches to Software Engineering*. Vol. 5503. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 417–420. DOI: 10.1007/978-3-642-00593-0_28.

Bruns, Daniel, Vladimir Klebanov, and Ina Schaefer (2011a). “Verification of Software Product Lines with Delta-Oriented Slicing”. In: *FoVeOOS*. Springer, pp. 61–75.

– (2011b). “Verification of Software Product Lines with Delta-Oriented Slicing”. In: *Formal Verification of Object-Oriented Software*. Vol. 6528. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 61–75. DOI: 10.1007/978-3-642-18070-5_5.

Bubel, Richard, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, Olaf Owe, Ina Schaefer, and Ingrid Chieh Yu (2016). “Proof Repositories for Compositional Verification of Evolving Software Systems”. In: *Transactions on Foundations for Mastering Change I*. Vol. 9960. Cham: Springer International Publishing, pp. 130–156. DOI: 10.1007/978-3-319-46508-1_8.

Cadar, Cristian and Martin Nowack (Dec. 2021). “KLEE Symbolic Execution Engine in 2019”. In: *International Journal on Software Tools for Technology Transfer* 23.6, pp. 867–870. DOI: 10.1007/s10009-020-00570-3.

Chalin, Patrice, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll (2006). "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2". In: *Formal Methods for Components and Objects*. Vol. 4111. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 342–363. DOI: 10.1007/11804192_16.

Chaudhuri, Kaustuv, Damien Doligez, Leslie Lamport, and Stephan Merz (2010). "Verifying Safety Properties with the TLA + Proof System". In: *Automated Reasoning*. Vol. 6173. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 142–148. DOI: 10.1007/978-3-642-14203-1_12.

Cheon, Yoonsik and Gary T. Leavens (2002). "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way". In: *ECOOP 2002 — Object-Oriented Programming*. Vol. 2374. Berlin, Heidelberg: Springer, pp. 231–255. DOI: 10.1007/3-540-47993-7_10.

Christakis, Maria, Peter Müller, and Valentin Wüstholz (2012). "Collaborative Verification and Testing with Explicit Assumptions". In: *FM 2012: Formal Methods*. Vol. 7436. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 132–146. DOI: 10.1007/978-3-642-32759-9_13.

Clarke, Edmund M., Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (2018). *Handbook of Model Checking*. Cham: Springer International Publishing. DOI: 10.1007/978-3-319-10575-8.

Classen, Andreas, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens (Feb. 2014). "Formal Semantics, Modular Specification, and Symbolic Verification of Product-Line Behaviour". In: *Science of Computer Programming* 80, pp. 416–439. DOI: 10.1016/j.scico.2013.09.019.

Classen, Andreas, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin (Aug. 2013). "Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking". In: *IEEE Transactions on Software Engineering* 39.8, pp. 1069–1089. DOI: 10.1109/TSE.2012.86.

Classen, Andreas, Patrick Heymans, and Pierre-Yves Schobbens (2008). "What's in a Feature: A Requirements Engineering Perspective". In: *Fundamental Approaches to Software Engineering*. Vol. 4961. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 16–30. DOI: 10.1007/978-3-540-78743-3_2.

Clifton, Curtis (2005). "A Design Discipline and Language Features for Modular Reasoning in Aspect-Oriented Programs". PhD thesis. Iowa State University.

Cok, David R. (July 2021). "JML and OpenJML for Java 16". In: *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*. Virtual Denmark: ACM, pp. 65–67. DOI: 10.1145/3464971.3468417.

Coq (1989/2025). *The Coq Proof Assistant*.

Cordy, Maxime, Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens (2013a). "Model Checking Adaptive Software with Featured Transition Systems". In: *Assurances for Self-Adaptive Systems*. Vol. 7740. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–29. DOI: 10.1007/978-3-642-36249-1_1.

Cordy, Maxime, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay (June 2012a). "Simulation-Based Abstractions for Software Product-Line Model Checking". In: *2012 34th International Conference on*

Software Engineering (ICSE). Zurich: IEEE, pp. 672–682. DOI: 10.1109/ICSE.2012.6227150.

Cordy, Maxime, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay (Sept. 2012b). "Behavioural Modelling and Verification of Real-Time Software Product Lines". In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. Salvador Brazil: ACM, pp. 66–75. DOI: 10.1145/2362536.2362549.

– (May 2013b). "Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features". In: *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, pp. 472–481. DOI: 10.1109/ICSE.2013.6606593.

Cuoq, Pascal, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski (2012). "Frama-C". In: *Software Engineering and Formal Methods*. Vol. 7504. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 233–247. DOI: 10.1007/978-3-642-33826-7_16.

Czech, Mike, Marie-Christine Jakobs, and Heike Wehrheim (2015). "Just Test What You Cannot Verify!" In: *Fundamental Approaches to Software Engineering*. Vol. 9033. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 100–114. DOI: 10.1007/978-3-662-46675-9_7.

Damiani, Ferruccio, Olaf Owe, Johan Dovland, Ina Schaefer, Einar Broch Johnsen, and Ingrid Chieh Yu (Sept. 2012). "A Transformational Proof System for Delta-Oriented Programming". In: *Proceedings of the 16th International Software Product Line Conference - Volume 2*. Salvador Brazil: ACM, pp. 53–60. DOI: 10.1145/2364412.2364422.

De Gouw, Stijn, Frank S. De Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel (Jan. 2019). "Verifying OpenJDK's Sort Method for Generic Collections". In: *Journal of Automated Reasoning* 62.1, pp. 93–126. DOI: 10.1007/s10817-017-9426-4.

De Gouw, Stijn, Jurriaan Rot, Frank S. De Boer, Richard Bubel, and Reiner Hähnle (2015). "OpenJDK's Java.Utils.Collection.Sort() Is Broken: The Good, the Bad and the Worst Case". In: *Computer Aided Verification*. Vol. 9206. Cham: Springer International Publishing, pp. 273–289. DOI: 10.1007/978-3-319-21690-4_16.

De Moura, Leonardo and Nikolaj Bjørner (2008). "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 4963. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

De Moura, Leonardo, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob Von Raumer (2015). "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25*. Vol. 9195. Cham: Springer International Publishing, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26.

DeMillo, R.A., R.J. Lipton, and F.G. Sayward (Apr. 1978). "Hints on Test Data Selection: Help for the Practicing Programmer". In: *Computer* 11.4, pp. 34–41. DOI: 10.1109/C-M.1978.218136.

Deng, Xianghua, Robby, and John Hatcliff (Sept. 2007). "Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems".

In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. Windsor: IEEE, pp. 3–12. DOI: 10.1109/TAIC.PART.2007.32.

Dhara, K.K. and G.T. Leavens (1996). “Forcing Behavioral Subtyping through Specification Inheritance”. In: *Proceedings of IEEE 18th International Conference on Software Engineering*. Berlin, Germany: IEEE Comput. Soc. Press, pp. 258–267. DOI: 10.1109/ICSE.1996.493421.

Dijkstra, Edsger W. (Aug. 1975). “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Communications of the ACM* 18.8, pp. 453–457. DOI: 10.1145/360933.360975.

Dijkstra, Edsger Wybe (1976). *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Englewood Cliffs, N.J: Prentice-Hall.

Dinca, Ionut, Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose (2011). “Test Data Generation for Event-B Models Using Genetic Algorithms”. In: *Software Engineering and Computer Systems*. Vol. 181. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 76–90. DOI: 10.1007/978-3-642-22203-0_7.

Edvardsson, Jon (1999). “A Survey on Automatic Test Data Generation”. In: *Proceedings of the 2nd Conference on Computer Science and Engineering*, pp. 21–28.

Engel, Christian and Reiner Hähnle (2007). “Generating Unit Tests from Formal Proofs”. In: *Tests and Proofs*. Vol. 4454. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 169–188. DOI: 10.1007/978-3-540-73770-4_10.

Erbsen, Andres, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala (Aug. 2020). “Simple High-Level Code For Cryptographic Arithmetic: With Proofs, Without Compromises”. In: *ACM SIGOPS Operating Systems Review* 54.1, pp. 23–30. DOI: 10.1145/3421473.3421477.

Ettinger, Ran (2021). “Lessons of Formal Program Design in Dafny”. In: *Formal Methods Teaching*. Vol. 13122. Cham: Springer International Publishing, pp. 84–100. DOI: 10.1007/978-3-030-91550-6_7.

Figueiredo, Eduardo and Nelio Cacho (2008). “Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability”. In: *Proceedings of the 13th International Conference on Software Engineering - ICSE ’08*. Leipzig, Germany: ACM Press, p. 261. DOI: 10.1145/1368088.1368124.

Filliâtre, Jean-Christophe and Claude Marché (2007). “The Why/Krakatoa/Caduceus Platform for Deductive Program Verification”. In: *Computer Aided Verification*. Springer, pp. 173–177.

Findler, Robert Bruce, Mario Latendresse, and Matthias Felleisen (Sept. 2001). “Behavioral Contracts and Behavioral Subtyping”. In: *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Vienna Austria: ACM, pp. 229–236. DOI: 10.1145/503209.503240.

Flaschenträger, Markus (2024). “Reusable Verification of Software Product Lines Using Proof Repositories”. Bachelor’s Thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology.

Furia, Carlo A., Martin Nordio, Nadia Polikarpova, and Julian Tschannen (Nov. 2017). "AutoProof: Auto-Active Functional Verification of Object-Oriented Programs". In: *International Journal on Software Tools for Technology Transfer* 19.6, pp. 697–716. DOI: 10.1007/s10009-016-0419-0.

Gacek, Critina and Michalis Anastasopoules (May 2001). "Implementing Product Line Variabilities". In: *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*. Toronto Ontario Canada: ACM, pp. 109–117. DOI: 10.1145/375212.375269.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. AddisonWesleyLongman.

Gentzen, Gerhard (1935). "Untersuchungen Über Das Logische Schließen. II". In: 39, pp. 405–431.

– (1964). "Investigations into Logical Deduction". In: 1, pp. 288–306.

Gondal, Ali, Michael Poppleton, and Michael Butler (2011). "Composing Event-B Specifications - Case-Study Experience". In: *Software Composition*. Vol. 6708. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 100–115. DOI: 10.1007/978-3-642-22045-6_7.

Gries, David (1981). *The Science of Programming*. Texts and Monographs in Computer Science. New York: Springer.

Gulwani, Sumit, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan (2010). *Component Based Synthesis Applied to Bitvector Programs*.

Hähnle, Reiner and Ina Schaefer (2012). "A Liskov Principle for Delta-Oriented Programming". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Vol. 7609. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 32–46. DOI: 10.1007/978-3-642-34026-0_4.

Hähnle, Reiner, Ina Schaefer, and Richard Bubel (2013). "Reuse in Software Verification by Abstract Method Calls". In: *Automated Deduction – CADE-24*. Vol. 7898. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 300–314. DOI: 10.1007/978-3-642-38574-2_21.

Hall, A. and R. Chapman (2002). "Correctness by Construction: Developing a Commercial Secure System". In: *IEEE Software* 19.1, pp. 18–25. DOI: 10.1109/52.976937.

Hall, Robert J. (Jan. 2005). "Fundamental Nonmodularity in Electronic Mail". In: *Automated Software Engineering* 12.1, pp. 41–79. DOI: 10.1023/B:AUSE.0000049208.84702.84.

Harel, David, Dexter Kozen, and Jerzy Tiuryn (Mar. 2001). "Dynamic Logic". In: *ACM SIGACT News* 32.1, pp. 66–69. DOI: 10.1145/568438.568456.

Hatcliff, John, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson (June 2012). "Behavioral Interface Specification Languages". In: *ACM Computing Surveys* 44.3, pp. 1–58. DOI: 10.1145/2187671.2187678.

Heisel, Maritta (1992). "Formalizing and Implementing Gries' Program Development Method in Dynamic Logic". In: *Science of Computer Programming* 18.1, pp. 107–137. DOI: 10.1016/0167-6423(92)90035-A.

Hiep, Hans Dieter A., Olaf Maathuis, Jinting Bian, Frank S. de Boer, Marko van Eekelen, and Stijn de Gouw (2020a). "Verifying OpenJDK's LinkedList Using

KeY". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12079 LNCS, pp. 217–234. DOI: 10.1007/978-3-030-45237-7_13.

Hiep, Hans-Dieter A., Jinting Bian, Frank S. De Boer, and Stijn De Gouw (2020b). "History-Based Specification and Verification of Java Collections in KeY". In: *Integrated Formal Methods*. Vol. 12546. Cham: Springer International Publishing, pp. 199–217. DOI: 10.1007/978-3-030-63461-2_11.

Hoare, C. A. R. (Oct. 1969). "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10, pp. 576–580. DOI: 10.1145/363235.363259.

– (1972). "Proof of Correctness of Data Representations". In: *Acta Informatica* 1.4, pp. 271–281. DOI: 10.1007/BF00289507.

Hoare, Tony (Jan. 2003). "The Verifying Compiler: A Grand Challenge for Computing Research". In: *Journal of the ACM* 50.1, pp. 63–69. DOI: 10.1145/602382.602403.

Igarashi, Atshushi, Benjamin Pierce, and Philip Wadler (Oct. 1999). "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM SIGPLAN Notices* 34.10, pp. 132–146. DOI: 10.1145/320385.320395.

ISO 26262-1:2018 (2018). *Road Vehicles — Functional Safety*.

Jackson, Daniel (2016). *Software Abstractions: Logic, Language, and Analysis*. Revised edition. Cambridge, Massachusetts London, England: The MIT Press.

Jacobs, Bart, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens (2011). "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods*. Vol. 6617. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4.

Jia, Yue and Mark Harman (Sept. 2011). "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5, pp. 649–678. DOI: 10.1109/TSE.2010.62.

Johnsen, Einar Broch, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen (2011). "ABS: A Core Language for Abstract Behavioral Specification". In: *Formal Methods for Components and Objects*. Vol. 6957. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 142–164. DOI: 10.1007/978-3-642-25271-6_8.

Kang, Kyo C., Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21. Software Engineering Institute.

Kästner, Christian, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger (Oct. 2011). "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. Portland Oregon USA: ACM, pp. 805–824. DOI: 10.1145/2048066.2048128.

Kelly, John C. (1997). "Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems Volume II: A Practitioner's Companion". In: Citeseer.

Khurshid, Sarfraz and Darko Marinov (Oct. 2004). "TestEra: Specification-Based Testing of Java Programs Using SAT". In: *Automated Software Engineering* 11.4, pp. 403–434. DOI: 10.1023/B:AUSE.0000038938.10589.b9.

Klabnik, Steve (2023). *The Rust Programming Language*. New York: No Starch Press.

Klebanov, Vladimir (2006). "Proof Reuse". In: *Verification of Object-Oriented Software. The KeY Approach*. Vol. 4334. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 507–529. DOI: 10.1007/978-3-540-69061-0_13.

Knüppel, Alexander, Stefan Krüger, Thomas Thüm, Richard Bubel, Sebastian Krieter, Eric Bodden, and Ina Schaefer (2020a). "Using Abstract Contracts for Verifying Evolving Features and Their Interactions". In: *Deductive Software Verification: Future Perspectives*. Vol. 12345. Cham: Springer International Publishing, pp. 122–148. DOI: 10.1007/978-3-030-64354-6_5.

Knüppel, Alexander, Tobias Runge, and Ina Schaefer (2020b). "Scaling Correctness-by-Construction". In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*. Vol. 12476. Cham: Springer International Publishing, pp. 187–207. DOI: 10.1007/978-3-030-61362-4_10.

Knüppel, Alexander, Leon Schaer, and Ina Schaefer (May 2021). "How Much Specification Is Enough? Mutation Analysis for Software Contracts". In: *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*. Madrid, Spain: IEEE, pp. 42–53. DOI: 10.1109/FormaliSE52586.2021.00011.

Knüppel, Alexander, Thomas Thüm, Carsten Pardylla, and Ina Schaefer (2018). "Experience Report on Formally Verifying Parts of OpenJDK's API with KeY". In: DOI: 10.48550/ARXIV.1811.10818.

Kourie, Derrick G. and Bruce W. Watson (2012). *The Correctness-by-Construction Approach to Programming*. Berlin, Heidelberg: Springer. DOI: 10.1007/978-3-642-27919-5.

Kovács, Laura and Andrei Voronkov (2013). "First-Order Theorem Proving and Vampire". In: *Computer Aided Verification*. Vol. 8044. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–35. DOI: 10.1007/978-3-642-39799-8_1.

Kuiter, Elias (2020). "Proof Repositories for Correct-by-Construction Software Product Lines". MA thesis. University of Magdeburg.

Leavens, Gary T. and Yoonsik Cheon (2006). *Design by Contract with JML*.

Leavens, Gary T. and Peter Muller (May 2007). "Information Hiding and Visibility in Interface Specifications". In: *29th International Conference on Software Engineering (ICSE'07)*. Minneapolis, MN: IEEE, pp. 385–395. DOI: 10.1109/ICSE.2007.44.

Leino, K. Rustan M. (2010). "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Vol. 6355. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

Leino, K. Rustan M., Peter Müller, and Jan Smans (2009). "Verification of Concurrent Programs with Chalice". In: *Foundations of Security Analysis and Design V*. Vol. 5705. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 195–222. DOI: 10.1007/978-3-642-03829-7_7.

Leino, K. Rustan M. and Greg Nelson (Sept. 2002). “Data Abstraction and Information Hiding”. In: *ACM Transactions on Programming Languages and Systems* 24.5, pp. 491–553. DOI: 10.1145/570886.570888.

Leitner, Andreas, Ilinca Ciupa, Bertrand Meyer, and Mark Howard (Jan. 2007). “Reconciling Manual and Automated Testing: The AutoTest Experience”. In: *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS’07)*. Waikoloa, HI: IEEE, 261a–261a. DOI: 10.1109/HICSS.2007.462.

Leroy, Xavier (July 2009). “Formal Verification of a Realistic Compiler”. In: *Communications of the ACM* 52.7, pp. 107–115. DOI: 10.1145/1538788.1538814.

Liskov, Barbara and John Guttag (1986). *Abstraction and Specification in Program Development*. MIT Press.

Liu, Jing, Josh Dehlinger, and Robyn Lutz (Nov. 2007). “Safety Analysis of Software Product Lines Using State-Based Modeling”. In: *Journal of Systems and Software* 80.11, pp. 1879–1892. DOI: 10.1016/j.jss.2007.01.047.

Ma, Yu-Seung, Jeff Offutt, and Yong-Rae Kwon (May 2006). “MuJava: A Mutation System for Java”. In: *Proceedings of the 28th International Conference on Software Engineering*. Shanghai China: ACM, pp. 827–830. DOI: 10.1145/1134285.1134425.

Manna, Zohar and Richard Waldinger (1980). “A Deductive Approach to Program Synthesis”. In: *ACM Transactions on Programming Languages and Systems* 2.1, pp. 90–121. DOI: 10.1145/357084.357090.

Martin, Robert C (2009). “Clean Code: A Handbook of Agile Software Craftsmanship”. In: *Pearson Education*. DOI: 10.1108/03684920910973252.

Meinicke, Jens, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake (2017). *Mastering Software Variability with FeatureIDE*. SpringerLink Bücher. Cham: Springer. DOI: 10.1007/978-3-319-61443-4.

Méry, Dominique and Neeraj Kumar Singh (2011). “Automatic Code Generation from Event-B Models”. In: *Proceedings of the Second Symposium on Information and Communication Technology - SoICT ’11*. Hanoi, Vietnam: ACM Press, p. 179. DOI: 10.1145/2069216.2069252.

Meyer, Bertrand (June 1988). “Eiffel: A Language and Environment for Software Engineering”. In: *Journal of Systems and Software* 8.3, pp. 199–246. DOI: 10.1016/0164-1212(88)90022-2.

– (Oct. 1992). “Applying ’Design by Contract’”. In: *Computer* 25.10, pp. 40–51. DOI: 10.1109/2.161279.

Molderez, Tim and Dirk Janssens (Mar. 2012). “Design by Contract for Aspects, by Aspects”. In: *Proceedings of the Eleventh Workshop on Foundations of Aspect-Oriented Languages*. Potsdam Germany: ACM, pp. 9–14. DOI: 10.1145/2162010.2162015.

– (2015). “Modular Reasoning in Aspect-Oriented Languages from a Substitution Perspective”. In: *Transactions on Aspect-Oriented Software Development XII*. Vol. 8989. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 3–59. DOI: 10.1007/978-3-662-46734-3_1.

Morgan, Carroll (1990). *Programming from Specifications*. Prentice Hall International Series in Computer Science. New York: Prentice Hall.

Mota, João, Marco Giunti, and António Ravara (Feb. 2023). *On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage*. DOI: 10.48550/arXiv.2209.05136. arXiv: 2209.05136 [cs].

Müller, Peter, Malte Schwerhoff, and Alexander J. Summers (2016). "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 9583. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2.

Oliveira, Marcel, Ana Cavalcanti, and Jim Woodcock (July 2003). "ArcAngel: A Tactic Language for Refinement". In: *Formal Aspects of Computing* 15.1, pp. 28–47. DOI: 10.1007/s00165-003-0003-8.

Peters, D.K. and D.L. Parnas (Mar. 1998). "Using Test Oracles Generated from Program Documentation". In: *IEEE Transactions on Software Engineering* 24.3, pp. 161–173. DOI: 10.1109/32.667877.

Pham, Thi-Kim-Dung (2017). "Development of Correct-by-Construction Software Using Product Lines". PhD thesis. Paris, CNAM.

Plath, Malte and Mark Ryan (Sept. 2001). "Feature Integration Using a Feature Construct". In: *Science of Computer Programming* 41.1, pp. 53–84. DOI: 10.1016/S0167-6423(00)00018-6.

Pohl, Klaus, Günter Böckle, and Frank J. van der Linden (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

Polikarpova, Nadia, Ivan Kuraj, and Armando Solar-Lezama (2016). "Program Synthesis from Polymorphic Refinement Types". In: *ACM SIGPLAN Notices* 51.6, pp. 522–538. DOI: 10.1145/2980983.2908093.

Post, Hendrik and Carsten Sinz (Sept. 2008). "Configuration Lifting: Verification Meets Software Configuration". In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. L'Aquila, Italy: IEEE, pp. 347–350. DOI: 10.1109/ASE.2008.45.

Prehofer, Christian (1997). "Feature-Oriented Programming: A Fresh Look at Objects". In: *ECOOP'97 — Object-Oriented Programming*. Vol. 1241. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 419–443. DOI: 10.1007/BFb0053389.

RTCA DO-178C (2011). *Software Considerations in Airborne Systems and Equipment Certification*.

Runge, Tobias, Alexander Kittelmann, Marco Servetto, Alex Potanin, and Ina Schaefer (2022). "Information Flow Control-by-Construction for an Object-Oriented Language". In: *Software Engineering and Formal Methods*. Vol. 13550. Cham: Springer International Publishing, pp. 209–226. DOI: 10.1007/978-3-031-17108-6_13.

Runge, Tobias, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick Kourie, and Bruce W. Watson (2019). "Tool Support for Correctness-by-Construction". In: *Fundamental Approaches to Software Engineering*. Vol. 11424. Cham: Springer, pp. 25–42. DOI: 10.1007/978-3-030-16722-6_2.

Runge, Tobias, Thomas Thüm, Loek Cleophas, Ina Schaefer, and Bruce W. Watson (2020). "Comparing Correctness-by-Construction with Post-Hoc Verification—A Qualitative User Study". In: *Formal Methods. FM 2019 International Workshops*

(Refine). Vol. 12233. Cham: Springer International Publishing, pp. 388–405. DOI: 10.1007/978-3-030-54997-8_25.

Savary, Aymerick, Marc Frappier, Michael Leuschel, and Jean-Louis Lanet (2015). “Model-Based Robustness Testing in Event-B Using Mutation”. In: *Software Engineering and Formal Methods*. Vol. 9276. Cham: Springer International Publishing, pp. 132–147. DOI: 10.1007/978-3-319-22969-0_10.

Scaletta, Marco, Reiner Hähnle, Dominic Steinhöfel, and Richard Bubel (Oct. 2021). “Delta-Based Verification of Software Product Families”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. Chicago IL USA: ACM, pp. 69–82. DOI: 10.1145/3486609.3487200.

Schaefer, Ina, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella (2010). “Delta-Oriented Programming of Software Product Lines”. In: *Software Product Lines: Going Beyond*. Vol. 6287. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 77–91. DOI: 10.1007/978-3-642-15579-6_6.

Schaefer, Ina, Lorenzo Bettini, and Ferruccio Damiani (Mar. 2011). “Compositional Type-Checking for Delta-Oriented Programming”. In: *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*. Porto de Galinhas Brazil: ACM, pp. 43–56. DOI: 10.1145/1960275.1960283.

Scholz, Wolfgang, Thomas Thüm, Sven Apel, and Christian Lengauer (Aug. 2011). “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*. Munich Germany: ACM, pp. 1–8. DOI: 10.1145/2019136.2019144.

Shahin, Ramy, Murad Akhundov, and Marsha Chechik (Mar. 2023). “Annotative Software Product Line Analysis Using Variability-Aware Datalog”. In: *IEEE Transactions on Software Engineering* 49.3, pp. 1323–1341. DOI: 10.1109/TSE.2022.3175752.

Shahin, Ramy and Marsha Chechik (Nov. 2020). “Automatic and Efficient Variability-Aware Lifting of Functional Programs”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA, pp. 1–27. DOI: 10.1145/3428225.

Shi, Jiangfan, Myra B. Cohen, and Matthew B. Dwyer (2012). “Integration Testing of Software Product Lines Using Compositional Symbolic Execution”. In: *Fundamental Approaches to Software Engineering*. Vol. 7212. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 270–284. DOI: 10.1007/978-3-642-28872-2_19.

Sistla, A. Prasad, Viktor Gyuris, and E. Allen Emerson (Apr. 2000). “SMC: A Symmetry-Based Model Checker for Verification of Safety and Liveness Properties”. In: *ACM Transactions on Software Engineering and Methodology* 9.2, pp. 133–166. DOI: 10.1145/350887.350891.

Smaragdakis, Yannis and Don Batory (Apr. 2002). “Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs”. In: *ACM Transactions on Software Engineering and Methodology* 11.2, pp. 215–255. DOI: 10.1145/505145.505148.

Smullyan, Raymond M. (1968). *First-Order Logic*. Berlin, Heidelberg: Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-86718-7.

Sorge, Jennifer, Michael Poppleton, and Michael Butler (2010). “A Basis for Feature-Oriented Modelling in Event-B”. In: *Abstract State Machines, Alloy, B and Z*. Vol. 5977. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 409–409. DOI: 10.1007/978-3-642-11811-1_42.

Stickel, Mark, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood (1994). “Deductive Composition of Astronomical Software from Subroutine Libraries”. In: *Automated Deduction — CADE-12*. Vol. 814. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 341–355. DOI: 10.1007/3-540-58156-1_24.

Strüber, Daniel, Sven Peldzsus, and Jan Jürjens (2018). “Taming Multi-Variability of Software Product Line Transformations”. In: *Fundamental Approaches to Software Engineering*. Vol. 10802. Cham: Springer International Publishing, pp. 337–355. DOI: 10.1007/978-3-319-89363-1_19.

Tartler, Reinhard, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat (Apr. 2011). “Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem”. In: *Proceedings of the Sixth Conference on Computer Systems*. Salzburg Austria: ACM, pp. 47–60. DOI: 10.1145/1966445.1966451.

Ter Beek, Maurice H., Loek Cleophas, Ina Schaefer, and Bruce W. Watson (2018). “X-by-Construction”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*. Vol. 11244. Cham: Springer International Publishing, pp. 359–364. DOI: 10.1007/978-3-030-03418-4_21.

Thüm, Thomas, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake (July 2014a). “A Classification and Survey of Analysis Strategies for Software Product Lines”. In: *ACM Computing Surveys* 47.1, pp. 1–45. DOI: 10.1145/2580950.

Thüm, Thomas, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund (2011a). “Abstract Features in Feature Modeling”. In: *SPLC*. IEEE, pp. 191–200.

Thüm, Thomas, Alexander Knüppel, Stefan Krüger, Stefanie Bolle, and Ina Schaefer (June 2019). “Feature-Oriented Contract Composition”. In: *Journal of Systems and Software* 152, pp. 83–107. DOI: 10.1016/j.jss.2019.01.044.

Thüm, Thomas, Jens Meinicke, Fabian Benduhn, Martin Hentschel, Alexander Von Rhein, and Gunter Saake (Sept. 2014b). “Potential Synergies of Theorem Proving and Model Checking for Software Product Lines”. In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*. Florence Italy: ACM, pp. 177–186. DOI: 10.1145/2648511.2648530.

Thüm, Thomas, Ina Schaefer, Sven Apel, and Martin Hentschel (Sept. 2012). “Family-Based Deductive Verification of Software Product Lines”. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. Dresden Germany: ACM, pp. 11–20. DOI: 10.1145/2371401.2371404.

Thüm, Thomas, Ina Schaefer, Martin Kuhlemann, and Sven Apel (Mar. 2011b). “Proof Composition for Deductive Verification of Software Product Lines”. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. Berlin, Germany: IEEE, pp. 270–277. DOI: 10.1109/ICSTW.2011.48.

Utting, Mark, Alexander Pretschner, and Bruno Legeard (Aug. 2012). “A Taxonomy of Model-based Testing Approaches”. In: *Software Testing, Verification and Reliability* 22.5, pp. 297–312. DOI: 10.1002/stvr.456.

Van Gurp, J., J. Bosch, and M. Svahnberg (2001). “On the Notion of Variability in Software Product Lines”. In: *Proceedings Working IEEE/IFIP Conference on Software Architecture*. Amsterdam, Netherlands: IEEE Comput. Soc, pp. 45–54. DOI: 10.1109/WICSA.2001.948406.

Von Rhein, Alexander, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel (Jan. 2016). “Variability Encoding: From Compile-Time to Load-Time Variability”. In: *Journal of Logical and Algebraic Methods in Programming* 85.1, pp. 125–145. DOI: 10.1016/j.jlamp.2015.06.007.

Watson, Bruce W., Derrick G. Kourie, Ina Schaefer, and Loek Cleophas (2016). “Correctness-by-Construction and Post-hoc Verification: A Marriage of Convenience?” In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*. Vol. 9952. Cham: Springer, pp. 730–748. DOI: 10.1007/978-3-319-47166-2_52.

Weidenbach, Christoph, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski (2009). “SPASS Version 3.5”. In: *Automated Deduction – CADE-22*. Vol. 5663. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 140–145. DOI: 10.1007/978-3-642-02959-2_10.

Weiß, Benjamin (2011). *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. Erscheinungsort nicht ermittelbar: KIT Scientific Publishing.

Weiss, David M., Paul C. Clements, Kyo Kang, and Charles Krueger (Aug. 2006). “Software Product Line Hall of Fame”. In: *Software Product Line Conference, International*. IEEE Computer Society, pp. 237–237. DOI: 10.1109/SPLC.2006.37.

Wenzel, Makarius, Lawrence C. Paulson, and Tobias Nipkow (2008). “The Isabelle Framework”. In: *Theorem Proving in Higher Order Logics*. Vol. 5170. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 33–38. DOI: 10.1007/978-3-540-71067-7_7.

Winskel, Glynn (1993). *The Formal Semantics of Programming Languages: An Introduction*. 5. printing. Foundations of Computing. Cambridge, Mass.: MIT Press.

Yilmaz, Cemal (May 2013). “Test Case-Aware Combinatorial Interaction Testing”. In: *IEEE Transactions on Software Engineering* 39.5, pp. 684–706. DOI: 10.1109/TSE.2012.65.

Zeyda, Frank, Marcel Oliveira, and Ana Cavalcanti (Dec. 2009). “Supporting ArcAngel in ProofPower”. In: *Electronic Notes in Theoretical Computer Science* 259, pp. 225–243. DOI: 10.1016/j.entcs.2009.12.027.

Zimmerman, Daniel M. and Rinkesh Nagmoti (2011). “JMLUnit: The Next Generation”. In: *Formal Verification of Object-Oriented Software*. Vol. 6528. Berlin, Heidelberg: Springer, pp. 183–197. DOI: 10.1007/978-3-642-18070-5_13.

Publications

This thesis is based on the following peer-reviewed publications.

Bordis, Tabea, Loek Cleophas, Alexander Kittelmann, Tobias Runge, Ina Schaefer, and Bruce 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. Cham: Springer International Publishing, pp. 80–104. DOI: 10.1007/978-3-031-08166-8_5.

Bordis, Tabea, Maximilian Kodetzki, Tobias Runge, and Ina Schaefer (2023a). "VarCorC: Developing Object-Oriented Software Product Lines Using Correctness-by-Construction". In: *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops*. Vol. 13765. Cham: Springer International Publishing, pp. 156–163. DOI: 10.1007/978-3-031-26236-4_13.

Bordis, Tabea, Maximilian Kodetzki, and Ina Schaefer (July 2024). "From Concept to Reality: Leveraging Correctness-by-Construction for Better Algorithm Design". In: *Computer* 57.7, pp. 113–119. DOI: 10.1109/MC.2024.3390948.

Bordis, Tabea, Tobias Runge, Fynn Demmler, and Ina Schaefer (2025 (Submitted)). "Improving Correctness-by-Construction Engineering by Increasing Levels of Correctness Guarantees". In: *International Journal on Software Tools for Technology Transfer*.

Bordis, Tabea, Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer (Feb. 2020a). "Variational Correctness-by-Construction". In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. Magdeburg Germany: ACM, pp. 1–9. DOI: 10.1145/3377024.3377038.

Bordis, Tabea, Tobias Runge, and Ina Schaefer (Nov. 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*. Virtual USA: ACM, pp. 22–34. DOI: 10.1145/3425898.3426959.

Bordis, Tabea, Tobias Runge, David Schultz, and Ina Schaefer (June 2022b). "Family-Based and Product-Based Development of Correct-by-Construction Software Product Lines". In: *Journal of Computer Languages* 70, p. 101119. DOI: 10.1016/j.cola.2022.101119.

Further peer-reviewed publications related to this thesis.

Bordis, Tabea and K. Rustan M. Leino (2024). “Free Facts: An Alternative to Inefficient Axioms in Dafny”. In: *Formal Methods*. Vol. 14933. Cham: Springer Nature Switzerland, pp. 151–169. DOI: 10.1007/978-3-031-71162-6_8.

Bordis, Tabea, Tobias Runge, Alexander Kittelmann, and Ina Schaefer (Apr. 2023b). “Correctness-by-Construction: An Overview of the CorC Ecosystem”. In: *ACM SIGAda Ada Letters* 42.2, pp. 75–78. DOI: 10.1145/3591335.3591343.

Kittelmann, Alexander, Tobias Runge, Tabea Bordis, and Ina Schaefer (2022). “Runtime Verification of Correct-by-Construction Driving Maneuvers”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*. Vol. 13701. Cham: Springer International Publishing, pp. 242–263. DOI: 10.1007/978-3-031-19849-6_15.

Kodetzki, Maximilian, Tabea Bordis, Michael Kirsten, and Ina Schaefer (2025a). “Towards AI-Assisted Correctness-by-Construction Software Development”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering Methodologies*. Vol. 15222. Cham: Springer Nature Switzerland, pp. 222–241. DOI: 10.1007/978-3-031-75387-9_14.

Kodetzki, Maximilian, Tabea Bordis, Alex Potanin, and Ina Schaefer (2025b). “X-by-Construction: Towards Ensuring Non-functional Properties in By-Construction Engineering”. In: *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! ’25)*. Singapore.

Kodetzki, Maximilian, Tabea Bordis, Tobias Runge, and Ina Schaefer (Feb. 2024). “Partial Proofs to Optimize Deductive Verification of Feature-Oriented Software Product Lines”. In: *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems*. Bern Switzerland: ACM, pp. 17–26. DOI: 10.1145/3634713.3634714.

Kuiter, Elias, Alexander Knüppel, Tabea Bordis, Tobias Runge, and Ina Schaefer (Feb. 2022). “Verification Strategies for Feature-Oriented Software Product Lines”. In: *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*. Florence Italy: ACM, pp. 1–9. DOI: 10.1145/3510466.3511272.

Rønneberg, Rasmus Carl, Tabea Bordis, Christopher Gerking, Asmae Heydari Tabar, and Ina Schaefer (2025). “Scaling Information Flow Control By-Construction to Component-based Software Architectures”. In: *45th International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Lille, France. DOI: 10.1007/978-3-031-95497-9_4.

Runge, Tobias, Tabea Bordis, Alex Potanin, Thomas Thüm, and Ina Schaefer (June 2023). “Flexible Correct-by-Construction Programming”. In: *Logical Methods in Computer Science* Volume 19, Issue 2, p. 10384. DOI: 10.46298/lmcs-19(2:16)2023.

Runge, Tobias, Tabea Bordis, Thomas Thüm, and Ina Schaefer (2021). “Teaching Correctness-by-Construction and Post-hoc Verification – The Online Experience”.

In: *Formal Methods Teaching*. Vol. 13122. Cham: Springer International Publishing, pp. 101–116. DOI: 10.1007/978-3-030-91550-6_8.