VarCorC: Developing Object-Oriented Software Product Lines Using Correctness-by-Construction

Tabea Bordis[1(✉)], Maximilian Kodetzki[2], Tobias Runge[1], and Ina Schaefer[1]

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{tabea.bordis,tobias.runge,ina.schaefer}@kit.edu
[2] TU Braunschweig, Braunschweig, Germany
m.kodetzki@tu-bs.de

Abstract. Functional correctness is an important concern, especially in the field of safety-critical systems. Correctness-by-Construction (CbC) is an incremental software development technique to create functionally correct programs guided by a formal specification. The specification is defined first, and then the program is incrementally created using a small set of refinement rules that define side conditions preserving the correctness. CbC is mostly used to create small algorithms. However, software in-field is often larger and more complex to meet the requirements of today's life. Therefore, our vision is to scale the applicability of CbC to larger scale software systems, like software product lines (SPLs). SPLs are one way to implement a whole product family by managed reuse. Advanced implementation techniques for SPLs rely on object-orientation and variability realization mechanisms on the source code level.

In this tool paper, we present our tool VarCorC which supports the development of correct SPLs using CbC including object-orientation and feature-oriented programming. We describe VarCorC from user-perspective and explain how it works internally. Additionally, we provide a feasibility evaluation of VarCorC on three case studies that are used as benchmarks in the field of product line verification.

Keywords: Correctness-by-Construction · Software product lines · Object-oriented programming · Program verification

1 Introduction

The demand for software in electronic devices is rapidly increasing, also including safety-critical applications like in the automotive, medical, or avionic field [13]. Correctness-by-Construction (CbC) as proposed by Dijkstra [9], Gries [10], or Kourie and Watson [12] gives a guarantee for functionally correct software which is crucial for safety-critical applications. CbC follows an incremental approach of program construction based on a formal specification in form of pre- and post-condition pairs. The specification is refined into an implementation using a set

P. Masci et al. (Eds.): SEFM 2022 Collocated Workshops, LNCS 13765, pp. 156–163, 2023.
https://doi.org/10.1007/978-3-031-26236-4_13

of refinement rules. To guarantee the correctness of these refinement steps, each rule defines specific side conditions for its applicability. In comparison to CbC, classical post-hoc verification offers an approach where a program is specified and verified *after* implementation. As a result, when using CbC errors are likely to be detected earlier in the design process [14]. CorC [18] is a tool that supports CbC to develop single algorithms. First evaluation results show decreased verification effort compared to post-hoc verification [6,18].

Our long-term vision is to make the construction of correct software using CbC applicable for large-scale systems, such as *software product lines* (SPLs). SPLs [17] enable the implementation of product families that share a common code base by managed reuse [8], therefore lowering costs and effort in producing custom-tailored software. The common and varying parts of an SPL are called *features*. The relationship of these features are modeled in *feature models* and variability realization mechanisms are used to implement their functionality. In the end, software variants can be created according to a certain selection of features. Many implementation techniques for SPLs, such as FeatureHouse [4] or DeltaJ [11], rely on *object-oriented design* since it is well suited to model large software systems. However, object-orientation poses some challenges for verification as fields can be globally accessed and concepts like inheritance increase the complexity of dependencies between classes. The complexity even increases for SPLs since variability is added to the code by variability realization mechanisms. Besides CbC as we pursue it, there are also other tools that implement different refinement-based approaches, such as Event-B [1] and its platform Rodin [2], ArcAngel [15], and SOCOS [5]. However, Event-B works on automata-based systems rather than on code and specifications and they all do not support the development of SPLs.

In this tool paper, we present VarCorC as an extension of CorC to develop object-oriented SPLs using CbC. In previous work, VarCorC has been developed from single variational methods [6], to feature-oriented SPLs with methods as simple procedures [7]. In this tool paper, we focus on the integration of object-orientation into VarCorC to enable the development of large-scale SPLs, since object-orientation allows for more complex projects and feature interactions over fields and objects. As specification, we use pre- and postconditions for methods and class invariants. Besides technical details of VarCorC, we also provide a workflow description from user-perspective to highlight VarCorC's usability features. Lastly, we present a short feasibility evaluation on three case studies.

2 The Development Process with VarCorC

In this section, we describe the development process in VarCorC as shown in Fig. 1 from the perspective of developer Alice. Alice develops an SPL that implements a bank account system and has already created a feature model ①. A feature model defines all features and their relationships in a tree structure. 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).

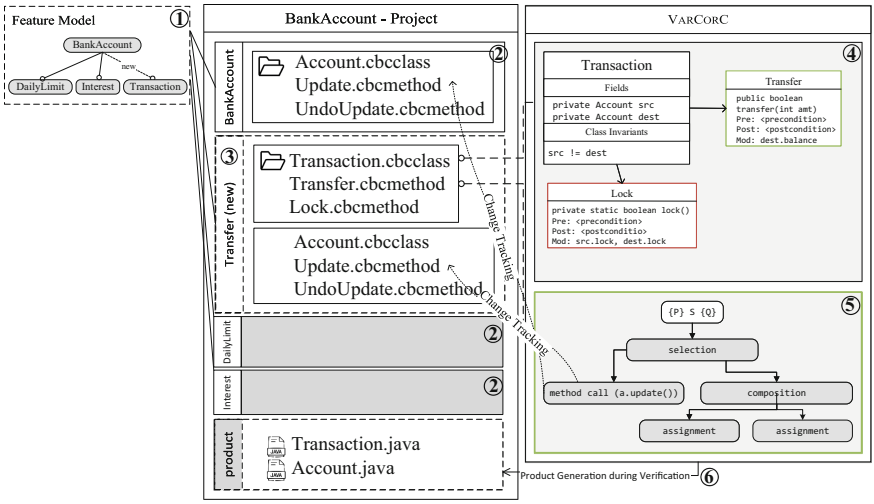

Fig. 1. Development process in VarCorC

Apart from the root feature *BankAccount*, all of the features are optional, which means that the user can select them individually. A valid selection of features is also called *feature configuration* and is used to form a software variant. Alice already implemented the features *BankAccount*, *Interest*, and *DailyLimit* in separate feature modules using feature-oriented programming (FOP) ②. A feature can add new classes or extend already existing classes by adding fields, class invariants, and methods or by refining existing methods. When a method is refined, its implementation is overridden with the option for reuse by using the FOP-specific keyword `original` to call the implementation of that method in another feature. Analogously, the specification of a method is overridden and the predicates `original_pre` and `original_post` can be used. In previous work [7], we already proposed an extension of CbC for original calls to implement methods in an SPL.

Alice now wants to add 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 by Kourie and Watson [12] and our refinement rules for method calls and original calls [7] which we display in Fig. 2. For example, to apply the assignment refinement rule a Hoare triple of the form `{P} S {Q}` with precondition

$\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$	*can be refined to*
1. Skip :	$\{\mathtt{P}\}$ *skip* $\{\mathtt{Q}\}$ *iff* $\mathtt{P}$ *implies* $\mathtt{Q}$
2. Assignment :	$\{\mathtt{P}\}\ \mathtt{x} := \mathtt{E}\ \{\mathtt{Q}\}$ *iff* $\mathtt{P}$ *implies* $\mathtt{Q}[\mathtt{x}\backslash\mathtt{E}]$
3. 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* $\{\mathtt{M}\}\ \mathtt{S}_2\ \{\mathtt{Q}\}$
4. Selection :	$\{\mathtt{P}\}$ **if** $\mathtt{G}_1 \rightarrow \mathtt{S}_1$ **elif** ... $\mathtt{G}_n \rightarrow \mathtt{S}_n$ **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* $\mathtt{i}$.
5. Repetition :	$\{\mathtt{P}\}$ **do** $[\mathtt{I}, \mathtt{V}]\ \mathtt{G} \rightarrow \mathtt{S}$ **od** $\{\mathtt{Q}\}$ *iff* ($\mathtt{P}$ *implies* $\mathtt{I}$) *and* ($\mathtt{I} \wedge \neg\mathtt{G}$ *implies* $\mathtt{Q}$) *and* $\{\mathtt{I} \wedge \mathtt{G}\}\ \mathtt{S}\ \{\mathtt{I}\}$ *and* $\{\mathtt{I} \wedge \mathtt{G} \wedge \mathtt{V}{=}\mathtt{V}_0\}\ \mathtt{S}\ \{\mathtt{I} \wedge 0{\leq}\mathtt{V} \wedge \mathtt{V}{<}\mathtt{V}_0\}$
6. Method Call :	$\{\mathtt{P}\}\ \mathtt{b} := \mathtt{m}(\mathtt{a}_1, ..., \mathtt{a}_n)\ \{\mathtt{Q}\}$ *with feature model* FM *and method* $\{\mathtt{P}'\}\ \mathtt{return\ r\ m}(\mathtt{param\ p}_1, ..., \mathtt{p}_n)\ \{\mathtt{Q}'\}$ *iff for all contracts* $c_i{=}\{\mathtt{P}'_i\}\ \mathtt{return\ r\ m}(\mathtt{param\ p}_1, .., \mathtt{p}_n)\ \{\mathtt{Q}'_i\}$ *composed over the set of feature configurations from* FM, *it holds that* $\mathtt{P}$ *implies* $\mathtt{P}'[\mathtt{p}_i\backslash\mathtt{a}_i]$ *and* $\mathtt{Q}'[\mathtt{p}_i^{\mathtt{old}}\backslash\mathtt{a}_i^{\mathtt{old}}, \mathtt{r}\backslash\mathtt{b}]$ *implies* $\mathtt{Q}$
7. Original Call :	$\{\mathtt{P}\}\ \mathtt{b} := \mathtt{original}(\mathtt{a}_1, ..., \mathtt{a}_n)\ \{\mathtt{Q}\}$ *with feature model* FM *and method* $\mathtt{m}$ *that is currently refined* $\{\mathtt{P}'\}\ \mathtt{return\ r\ m}(\mathtt{param\ p}_1, ..., \mathtt{p}_n)\ \{\mathtt{Q}'\}$ *iff for all contracts* c_i $c_i{=}\{\mathtt{P}'_i\}\ \mathtt{return\ r\ m}(\mathtt{param\ p}_1, .., \mathtt{p}_n)\ \{\mathtt{Q}'_i\}$ *composed over the set of feature configurations from* FM, *it holds that* $\mathtt{P}$ *implies* $\mathtt{P}'_i[\mathtt{p}_j\backslash\mathtt{a}_j]$ *and* $\mathtt{Q}'_i[\mathtt{p}_j^{\mathtt{old}}\backslash\mathtt{a}_j^{\mathtt{old}}, \mathtt{r}\backslash\mathtt{b}]$ *implies* $\mathtt{Q}$

Fig. 2. List of refinement rules in correctness-by-construction [12] and method call and original call refinement rule [7]

P, postcondition Q and abstract statement S can be refined to an assignment x := E with x being a variable and E an expression of the same type or subtype if and only if the side condition that precondition P implies postcondition Q where variable x has been replaced by expression is fulfilled. All of the listed refinement rules are implemented in VarCorC. 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 [3]. Therefore, the method under development is guaranteed to be correct.

During this verification process, all variants of a method according to the feature model are generated into Java classes ⑥. This has the advantage that (1) Alice can export correct code developed with VarCorC into other projects and (2) Alice can call externally implemented code in VarCorC when placed in these classes. As a result, Alice can decide about the degree of using CbC as opposed to using Java verified with a different tool or checked with testing.

One of VarCorC's main usability features provides Alice with an overview on the verification status of all methods in the SPL and the traceability of errors

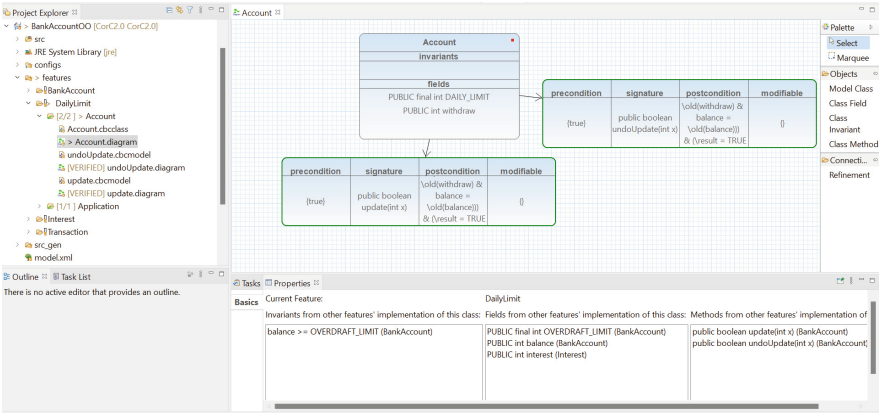

Fig. 3. Screenshot of VarCorC with class `Account` in feature *DailyLimit*

down to one refinement step. The first is enhanced by 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.

3 Object-Oriented Software Product Lines in VarCorC

In this section, we give implementation details for our tool VarCorC[1] which is an open-source Eclipse plug-in supporting the development of object-oriented SPLs using CbC and FOP. VarCorC captures the CbC structure of methods and classes through a meta-model modeled with Eclipse Modeling Framework.[2] The graphical editor visualizes the underlying meta-model in a tree-like structure for methods and UML-like class diagrams.

In Fig. 3, we show a screenshot of VarCorC with class `Account` in feature *DailyLimit*. The project structure consists of a feature model, feature

[1] VarCorC implements SPL development using CbC and is part of the tool CorC: https://github.com/TUBS-ISF/CorC.

[2] https://eclipse.org/emf/.

Table 1. Metrics of the case studies

Case study	Features	Classes	Methods	Original calls
BankAccount [21]	4	3	10	5
IntegerList [19]	5	1	5	2
Elevator [16]	5	4	35	5

modules, and class folders. The cbcclass and cbcmethod files are split into a *<methodName>\<classname>.diagram* file, which contains the graphical information, and a *<methodName>.cbcmodel\<classname>.cbcclass* file which is an instance of the corresponding meta-model. The *src-gen* folder contains generated Java classes, which store composed software variants for the 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 on class invariants, fields, and methods of class `Account` in other features.

In the center of Fig. 3, 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 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, every variant has to be correct. In FOP, each variant can have a different set of classes, classes can have a different sets of fields, class invariants, and methods, and methods can have different implementations and specifications. We use a product-based approach [20] for showing correctness. Once the verification of a refinement step in a method is triggered, all valid feature configurations are calculated such that original calls can be resolved. For each configuration, the corresponding variant in form of Java classes is generated. At the same time, a proof file is created which contains the side condition of the CbC refinement step. If all proofs are successful, the statement is considered to be correct.

Evaluation. We evaluate VarCorC regarding feasibility by implementing three case studies, namely *BankAccount* [21], *IntegerList* [19], and *Elevator* [16]. All case studies have already been used as benchmarks in SPL verification.[3] In Table 1, we show metrics for the case studies. The *BankAccount* SPL implements basic functions of a bank account and has been used throughout this paper as an example. The *IntegerList* SPL implements a list of integers with add and sort operations. The third case study, *Elevator*, implements basic functions of an elevator, such as the movement and entering and leaving of persons. We transferred the case studies into the object-oriented structure as introduced in this paper. For every class, we created cbcclass files with fields and class invariants. All methods are verified individually for all valid feature configurations in VarCorC, therefore showing correctness of the whole SPL.

[3] Case studies and VarCorC: https://github.com/TUBS-ISF/CorC.

4 Conclusion

We believe that the specification-first, refinement-based approach of CbC increases the awareness of correctness when developing safety-critical software in today's engineered world. Until recently, CbC has only been used for independent algorithms. Therefore, we presented our tool VarCorC which enables program development with CbC for object-oriented SPLs. We showed, how we include object-orientation into CbC and highlighted usability features of VarCorC that streamline the development of SPLs. Currently, VarCorC relies on a product-based approach limiting its scalability. Therefore, in future work we want to experiment with more efficient approaches, such as family-based verification.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transfer **12**(6), 447–466 (2010). https://doi.org/10.1007/s10009-010-0145-y
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification - The KeY Book. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
4. Apel, S., Kästner, C., Lengauer, C.: Language-independent and automated software composition: the FeatureHouse experience. IEEE Trans. Softw. Eng. **39**(1), 63–79 (2013)
5. Back, R.-J., Eriksson, J., Myreen, M.: Testing and verifying invariant based programs in the SOCOS environment. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 61–78. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73770-4_4
6. Bordis, T., Runge, T., Knüppel, A., Thüm, T., Schaefer, I.: Variational correctness-by-construction. In: Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems, pp. 1–9 (2020)
7. Bordis, T., Runge, T., Schaefer, I.: Correctness-by-construction for feature-oriented software product lines. In: International Conference on Generative Programming: Concepts and Experiences, pp. 22–34 (2020)
8. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Citeseer (2000)
9. Dijkstra, E.W.: A Discipline of Programming, 1st edn. Prentice Hall PTR (1976)
10. Gries, D.: The Science of Programming, 1st edn. Springer, New York (1981). https://doi.org/10.1007/978-1-4612-5983-1
11. Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Damiani, F.: DeltaJ 1.5: delta-oriented programming for Java 1.5. In: International Conference on Principles and Practices of Programming on the Java Platform, pp. 63–74 (2014)
12. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27919-5

13. Liu, J., Dehlinger, J., Lutz, R.: Safety analysis of software product lines using state-based modeling. J. Syst. Softw. **80**(11), 1879–1892 (2007)
14. Meyer, B.: Applying design by contract. Computer **25**(10), 40–51 (1992)
15. Oliveira, M., Cavalcanti, A., Woodcock, J.: ArcAngel: a tactic language for refinement. Formal Aspects Comput. **15**, 28–47 (2003). https://doi.org/10.1007/s00165-003-0003-8
16. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. **41**(1), 53–84 (2001)
17. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005). https://doi.org/10.1007/3-540-28901-1
18. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool support for correctness-by-construction. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 25–42. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_2
19. Scholz, W., Thüm, T., Apel, S., Lengauer, C.: Automatic detection of feature interactions using the Java modeling language: an experience report. In: International Software Product Line Conference (2011)
20. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. **47**(1), 1–45 (2014)
21. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-based deductive verification of software product lines. In: International Conference on Generative Programming and Component Engineering (2012)