From concurrent state machines to reliable multi-threaded Java code

From Concurrent State Machines to Reliable Multi-threaded Java Code

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. F.P.T. Baaijens, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 12 april 2018 om 16.00 uur

door

Dan Zhang

geboren te Henan, China

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. J.J. Lukkien
promotor:	prof.dr. M.G.J. van den Brand
copromotoren:	dr. R. Kuiper
	dr. D. Bosnacki
leden:	prof.dr.ir. L.M.G. Feijs
	prof.dr. M. Huisman (Universiteit Twente)
	prof.Dr. S. Leue (Universität Konstanz)
	prof.dr. P.-E. Moreau (Université de Lorraine)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

From Concurrent State Machines to Reliable Multi-threaded Java Code

Dan Zhang

Promotor:	prof.dr. M.G.J. van den Brand (Eindhoven University of Technology)
Copromotoren:	dr. R. Kuiper (Eindhoven University of Technology)
	dr. D. Bosnacki (Eindhoven University of Technology)

Additional members of the reading committee:

prof.dr.ir. L.M.G. Feijs	(Eindhoven University of Technology)
prof.dr. M. Huisman	(University of Twente)
prof.Dr. S. Leue	(Universität Konstanz)
prof.dr. P.-E. Moreau	(Université de Lorraine)

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
IPA dissertation series 2018-05.

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-4475-2

Cover design: The cover and layout was designed by Dan Zhang and the illustration was designed and painted by José Brands.

Printed by Gildeprint

Acknowledgements

The journey of pursuing my PhD study has been a life changing event for me. Whenever looking back upon those good days, I am always full of gratitude and appreciation to all those people who have brought me growth and development. This dissertation is no exception and I would not have succeeded without the help and support from all of you.

First and foremost I want to express my heartfelt thanks to my supervisor Prof. Mark van den Brand who gave me the opportunity to come to the Netherlands and work on this project. It has been an honor to be you PhD student. Mark, you are always able to guide me with your high-level overview on my topic. I remember so vividly the optimism and passion that you inspired me deeply at every moment of my PhD. Especially in the hard times of my PhD study, your trust and continuous encouragements helped me to move on. Your advice on both research as well as on my career have been priceless.

My special thanks go to my daily supervisors Dr. Ruurd Kuiper and Dr. Dragan Bosnacki for guiding me through my PhD journey. Dear Ruurd, you are always full of patience and ready to help me at any time I need. I cannot count how many times I knocked your office door for having discussions about my work or sharing stories about my life. You always managed to help me sort my research thinking out in a structural way using your strong scientific advice and knowledge. Thanks so much for your strong supports on helping me overcome all difficulties during my PhD. Dear Dragan, I really appreciate that you agreed to be a member of my supervising team when I started my PhD journey. At the beginning of my study you helped to guide me to dive into the ocean of model driven technologies, theorem proving and model checking etc. Thank you very much for spending time on many insightful discussions with me. I extremely enjoy our daily discussions about research and life, from which you often give me a more intuitive view of understanding and explaining things.

I would also like to express my deep gratitude to my PhD supporting team members Dr. Anton Wijs and Dr. Kees Huizing. Dear Anton, it is a great pleasure to work with you and I really appreciate your valuable feedbacks and fruitful discussions during the weekly meetings in the last four years. Your support and experience gave me a lot of confidence to finish this thesis. I am also very grateful for the trust and support from your and Prof. Marieke Huisman on my new postdoc project. Dear Kees, thanks a lot for joining my supervising team when I was searching an expert on Java language. You are always able to think outside the box to help me improve the Java translation presented in

this thesis. I really appreciate all the thorough discussions we had during last four years.

Throughout my PhD study I have collaborations with many people. Their supports contributed a lot to this dissertation. I would like to thank Dr. Bart Jacobs (Katholieke Universiteit Leuven), Dr. Luc Engelen, Sybren Roede, Philippe Denissen and Maciej Wiłkowski. My PhD work builds on Sybren Roede's master project (Suzana Andova was an inspiration to that). Thanks for his help in the initial stages. My special thanks go to Bart Jacobs for guiding me using the VeriFast tool for the verification presented in this thesis. Your valuable comments on the Java implementation helped me to make the current Java implementation simpler and more elegant. I enjoyed and learned a lot from the discussions on my research work.

Furthermore, I would like to extend my sincere gratitude to the members of my reading committee: prof.dr.ir. L.M.G. Feijs from Eindhoven University of Technology, prof.dr. M. Huisman form University of Twente, prof.Dr. S. Leue from Universität Konstanz and prof.dr. P.-E. Moreau from Université de Lorraine. I am very grateful for your valuable comments on my thesis draft and your participation in my PhD promotion. I would also like to express my deep gratitude to prof.dr. J.J. Lukkien for chairing my PhD defense ceremony.

To all my colleagues in the MDSE group, thanks for your kind supports, collaborations and enjoyable talking during my PhD. Especially, I would like to thank Luna (Yaping Luo) for introducing the Software Engineering and Technology group (SET) to me and helping me integrate into this group. Thank you very much for your great encouragement at the beginning of my PhD study and your continuous help during the past four years. My special thank also goes to Margje Mommers-Lenders for your continuous assistance for my work and your great patience to allow me share my happiness and sadness with you. I would like to convey my gratitude to Sander de Putter for the delightful experiences about 'work to live' during last four years. I also want to thank my former officemates: Ana-Maria Şutîi, Ulyana Tikhonova, Yanja Dajsuren, Neda Noroozi, Felipe Ebert and current officemates Weslley Silva Torres, Saurab Rajkarnikar and Gema Rodriguez Perez for having the joyful office life together. It was always a great fun to talk with all of you about science, cultures and foods. I also appreciate the rest of my current and previous colleagues, especially (in no particular order) Fei Yang, Thomas Neele, Mahmoud Talebi, Önder Babur, Kousar Aslam, Priyanka Karkhanis, Saneeth Kochanthara, Yuexu Chen, Raquel Alvarez Ramirez, Mauricio Verano Merino, Rodin Aarssen, Miguel Botto Tobar, Arash Khabbaz Saberi, Josh Mengerink, Frank Peter, Rob Faessen, Omar Alzuhaibi, Bogdan Vasilescu, Alexander Fedotov, Gerard Zwaan, Ramon Schiffelers, Alexander Serebrenik, Loek Cleophas, Jurgen Vinju, Tom Verhoeff, Tim Willemse, Erik de Vink, Hans Zantema, Julien Schmaltz, Jan Friso Groote and others.

During my PhD life, I participated in several classes which made my life much more enjoyable and fulfilling. Luckily, from those I met many nice people and made very sincere friends. I am very much grateful to Wieger Wesselink for your continuous understanding and encouragements that help me a lot to gain my confidence to go through the hard times of my PhD study. Thanks so much for offering me the English and Dutch classes in numerous evenings to improve my languages skills. I also appreciate your evening yoga class in the sports center which really helps me a lot to find the balance between life and work. I would also extend my gratitude to Ion Barosan for sharing your life experiences with me after your meditation class. I was quite impressed by the state diagram of my daily life drawn by you during one discussion. This state diagram gives me hints how to

become more positive during my daily life. Thank you very much for offering such relaxing and open meditation course to students.

Of course, my life in Eindhoven would not be so colorful without many friends here. Thanks to Brishna Nader for sharing delights of life, doing yoga and excursions together in the past years. Thanks to Hélène Arts and Mónica Fernandes for sharing your life experiences with me. In addition, I also want to thank Xixi Lu, Yonghui Li, Yingchao Cui, Weidong Zhang, Ruisheng Su, Yuan Chen, Zizheng Cao, Wenjie Bai, Ying Zhao, Bilin Han, Haitao Xing, Valcho Dimitrov and others. It was a great pleasure to enjoy lots of joy and fun with you.

The illustration of my thesis cover was designed and painted by José Brands. Thank you very much José for the brainstorm discussions about arts and life. I enjoyed the tours in Van Abbemuseum and bibliotheek in Eindhoven. Also, I was quite impressed by the exhibition in Lievendael, which wakes up my talent about painting. Thanks a lot for your time and efforts.

It took some time to finish this dissertation after my PhD study ended, and I thank Prof. Marieke Huisman and Dr. Anton Wijs for allowing me to work on it during my employment at University of Twente. Thank you very much for your understanding and concerns to let me regain confidence in my life and research.

Finally, I would like to express my deepest gratitude to my family. Thanks to my parents, sisters, brother, and my nieces and nephews for your unconditional love and consistent supports on my journey. Thanks to my husband Jiong for sharing all the happiness with me and supporting me to get through all the hard times. I really appreciate your understanding and great patience all along. The support from my family give me the greatest power to move on. Thanks for everything and now I am fully ready for challenges ahead.

Dan Zhang
Eindhoven, February 2018

Table of Contents

Chapter 1

Introduction

1.1 Background

The rise in popularity of modern parallel computing hardware, such as multi-core processors [10, 72, 83] and graphics processing units (GPUs) [25, 26, 52, 78, 80], has increased the demand for concurrent software that utilizes the available resources in an optimal way thus pushing the computational boundaries in many fields significantly. However, concurrent programs are usually difficult to develop. Due to the complexity of concurrency, such programs are more prone to containing errors, such as data races and deadlocks, than sequential programs. Some of these errors are hard to reason about because of their notorious non-deterministic characteristic. To make the development of correctly functioning parallel software easier, we conduct our research on Model-Driven Software Engineering (MDSE) techniques [88], in which the intended functionality is first modeled using a domain-specific modeling language (DSML) [50]. Next, software code is automatically generated via a model-to-code transformation, which prevents errors in the implementations that result from misinterpretations of designs of software systems.

MDSE, combining DSMLs with model transformations, is gaining popularity as a methodology for developing software in an efficient way. The methodology aims at dealing with the increasing software complexity and improving productivity and quality by raising the level of abstraction, using modeling techniques, model transformation, and code generation. DSMLs are modeling languages that offer, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [104]. Constructing models with DSMLs enables developers to deal with difficult aspects at a higher, less complex and more intuitive level of abstraction. As the domain concepts provided by DSMLs are typically not computing-oriented, domain-specific models are not directly usable for automatic execution [103]. For that purpose, model transformations are employed to transform domain-specific models to different, computing-oriented models [89]. Starting with an initial model written in a DSML, other artifacts such as additional models, source code and test scripts can be produced via a chain of model transformations. By shifting the focus from code to models, MDSE allows

to tackle defects of the software in the modeling phase. Resolving errors in the early stages of the software development process reduces the costs and increases the reliability of the end product.

One challenging task in MDSE is to produce correct and high quality code from high-level descriptions of models via model-to-code transformations. The challenging issues mainly originate in the lack of correspondence between model-oriented primitives in domain-specific modeling languages and their counterparts in programming languages. Moreover, DSMLs are often not designed to address all details needed when generating code, due to their high level of abstraction, so additional decisions have to be made in this step. Furthermore, as with other software development artifacts, model-to-code transformations and code produced by them may not be free of errors.

We use two main ideas to make code generation feasible. The first one is to make a distinction between generic DSML concepts used in a model and the part of the model that describes the specific behavior of a system. By doing this, we can improve the understandability, modifiability as well as re-usability of the code produced by a model-to-code transformation. The generic parts of models are the same for every model described in the same DSML; they are transformed, once, into generic code. Aspects that are specific for concrete models are transformed into specific code for each such model. To enable combining generic with specific code, a first kind of modularity is required, namely between the generic and the specific code.

The second idea is to treat each individual part of the generic code in a modular way. For each of the modeling language constructs there is a corresponding part in the generic code that provides its behavior. We call such an individual part a *construct* of the generic code, a name we stick to in the rest of the thesis. These constructs are designed to be modular, which provides several benefits. First and foremost, modularity enables the translation to be compositional at the level of modeling language constructs. Second, modification of some constructs of the generic code will then not affect the transformation or other constructs of the generic code and modularity facilitates reuse of the existing parts when the need for new target platforms arises. Third, each construct can then be understood in isolation, which benefits the overall understanding of the transformation. Lastly, the specification and verification of each construct can be modularized accordingly. This is the second, more fine grained, kind of modularity: between individual constructs of the generic code.

It turns out that providing the second modularity to a large extent provides the first one. The reason for this is that generic and specific code are connected at the level of individual constructs.

To ensure the correctness of produced code and transformations themselves, a formal specification logic and proof support is required. Verifying a transformation that transforms models written in a DSML to executable code in a programming language is fundamentally more complex than verifying an individual program. In particular, this requires semantic conformance between the model and the generated code. For models in the area of safety-critical concurrent systems, the main complication to guarantee this equivalence involves the potential of threads to non-deterministically interact with each other. This complication makes a formal proof difficult and time-consuming.

To make the complexity of verification of model-to-code transformations manageable, we specify the constructs in the generic code in a modular manner, i.e., supplying environment information in the specifications, foremost about the other generic constructs, but also

about the manner the specific code is intended to use the generic code. The benefits of such an approach are that it will scale better than a monolithic approach. Once a construct has been specified, we can abstract away its implementation details when verifying properties of other constructs. It is then also easier to reason about properties of the complete system.

Since verifying the correctness of the generic code is the main focus of this thesis, we therefore start by identifying modules within the generic code, then add modular specifications of these and then perform modular verification.

The techniques for the verification of traditional software artifacts, such as testing, model checking and theorem proving, have been investigated in the past to verify the correctness of model-to-code transformations [84]. Testing primarily focuses on producing meaningful test cases that inform about the possible executions of programs transformed from models. This approach helps to find bugs in the model transformations but there is no guarantee that all the errors are detected. Therefore, we use testing to efficiently assess potential weak points during the development of some parts of our generic code but aim for complement verification of the generic code. Model checking [18] achieves completeness for programs that have essentially a finite number of states, as it explores the entire state space of a program under all possible input conditions. Sulzmann et al. [95] apply the Spin model checker to verify whether C code generated from a high-level DSML description is correct. Staats et al. [92] use software model checking (the ANSI-C Bounded Model Checker) to check whether particular functional properties expressed in LTL are preserved by a transformation from Simulink models to C code. Ab. Rahim et al. [8] use a similar approach to verify a transformation from UML state machines to Java code. The main obstacle that model checking faces is the state explosion problem [32]. The number of global states of a concurrent system with multiple processes can be enormous or even infinite. In such cases, model checking is less suitable, although in recent years, advances have been made in exploring large state spaces [36,47,60,73,79,108,109,112,113]. However, as model checking usually targets closed systems, it is hard to verify the correctness of a component separated from its environment. Therefore, for verifying constructs of generic code in this thesis, we did not opt for model checking.

As in model checking, theorem proving mathematically proves the correctness of a program with respect to a mathematical formal specification. However, unlike model checking, verification by theorem proving is not limited by the size of state spaces of programs to be verified. This is because the assertions in theorem proving describe possibly infinitely many state values in a finite manner. Additionally, theorem proving allows for verifying each generic component separately in a modular way. In [22], a formal verification using the Isabelle/HOL theorem prover is presented of a concrete algorithm that generates Java code from UML Statecharts. It is shown that the source UML model and the generated Java code are bisimilar. In [93], a Java code generation framework based on the transformation language QVT is presented. The theorem prover KIV [19] is used to prove security properties and syntactic correctness of generated Java code. In [38, 39], annotations are generated together with code to assist automatic theorem proving. However, one of the major issues of these works is the scalability and complexity of the proof when the transformations are applied on more complex models.

This issue can be addressed by using the approach mentioned above, i.e., verifying the code in a modular way. Each construct and its implementation is independently specifiable and verifiable. Moreover, specifications for constructs of the generic code can be directly

used when verifying the specific code. As a result, the verification of large software systems can be decomposed into subproblems of manageable complexity.

We target the automated generation of multi-threaded Java programs from models. To verify the generic code in a modular way, we therefore firstly need a verification approach that supports object-oriented modularity, e.g., the formal proof system presented in [61] supports the verification and derivation of object-oriented sequential programs. Secondly, parallelism should be supported by the verification approach (e.g, the Owicki and Gries method [82] provides this). The VeriFast [91] tool we use is a program verifier which supports verification for object-oriented as well as multi-threaded Java programs. To achieve this, VeriFast uses as specification logic separation logic [81, 86].

1.2 Setting the Context

In the current thesis, we consider a restricted, yet representative MDSE work-flow using a DSML called the Simple Language of Communicating Objects (SLCO) [41, 103]. SLCO is defined for modeling complex embedded concurrent systems in which state machines communicate via either shared variables or explicit message passing over channels. A number of model-to-model transformations based on SLCO models and their quality are investigated in [12,41,102,103]. The work in [103] is concerned with the quality of definitions of model-to-model transformations, whereas the work in [12, 41, 102] focuses on the quality of the process of transforming a source model to a target model and its correctness in particular. As with model-to-model transformations [35, 37, 41, 44, 103, 107, 110, 111], the quality of model-to-code transformations, in particular their correctness, needs to be investigated. To this end, we focus on how to generate reliable software from models in the context of a framework that transforms SLCO models to parallel programs in Java via model-to-code transformation.

The transformation chain of SLCO models consists of multiple steps, as shown in Figure 1.1. The left dashed rectangle depicts a fine-grained sequence of model-to-model transformations which is used to deal with differences between SLCO and its target languages. This fine-grained sequence of transformations adds functional details in each refinement step and therefore a number of intermediate SLCO models, such as M^2_{SLCO}, M^3_{SLCO} and M^N_{SLCO}, are introduced. After all functional details have been added, the last step from an SLCO model to executable code needs to be applied, which is a model-to-code transformation (M2C), as shown in the solid rectangle in the figure. For the construction of parallel software in the last step, we target multi-threaded Java code. Although we treat code generation as one step in the model transformation chain, in general this could be done in multiple steps, for instance using the Tom language [16].

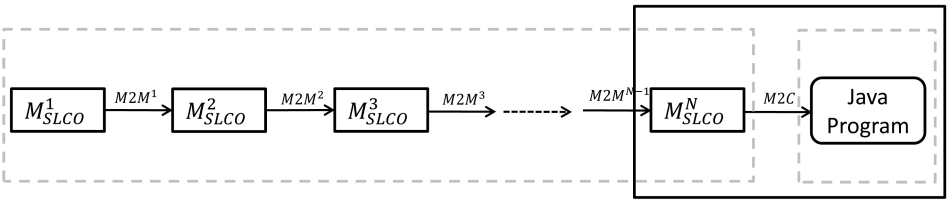

Figure 1.1: SLCO models to Java code transformation architecture.

To successfully implement the model-to-code transformation and generate reliable code, a number of problems need to be solved.

1.3 Problem Statement

As DSMLs focus on high-level domain-specific abstractions, they are often not expressive enough to address all implementation details regarding target platforms. Moreover, a target programming language may not have suitable constructs to directly implement all concepts of a modeling language. This leaves choices and thereby challenges to developers. For instance, synchronization mechanisms needed for shared data structures, such as variables and channels in SLCO models, are not addressed at the model level, whereas at the code level there are many options to implement them, e.g., using fine-grained locking or coarse-grained locking mechanisms. Compared to coarse-grained locking mechanisms, fine-grained ones can improve the performance of concurrent programs. However, they are more complicated to implement and are more prone to errors like deadlocks. Therefore, one of the prerequisites for generating reliable parallel application programs from models with high-level descriptions is that developers need to identify corresponding challenges and choices before designing the entire application's architecture and their algorithms for concurrent behavior.

Once challenges and choices are identified, the gaps between DSMLs and target programming languages should be bridged. Quality aspects, like modularity and efficiency which improve the quality of model-to-code transformations, also need to be taken into account, which imposes additional challenges. The challenges and choices force developers to think carefully about the interplay of parallel activities between multiple threads. The reason for this is that the interactions have to be in line with the functional specification given in the DSMLs when developing model-to-code transformations.

After model-to-code transformations are implemented, a question that naturally arises is how to guarantee that functional properties of the input models are preserved in the generated code. In particular, this requires semantic conformance between the model and the generated code. To ensure this, a formal specification of the functional properties and verification of their preservation are required.

As previously mentioned, safety-critical concurrent systems can be complex. To deal with this complexity, instead of interpreting models and the corresponding implementations with all their features monolithically, a software engineer can simplify implementations by focusing on some features of constructs. Such a simplification also facilitates the verification of the implementation. To improve the extensibility of the simplified implementations, each construct should be treated as a separate module, which requires modular verification.

An aspect not considered at the model level is the possible occurrence of exceptions during the execution of the program. Abnormal terminations caused by exceptions may lead to critical issues for parallel programs, such as safety violations caused by inconsistent data structures and deadlocks. To address this, existing handling mechanisms in programming languages should be investigated to improve the robustness of the produced code. To ensure that the handling mechanism leads to a correct implementation for any model it is applied to, the correctness of the handling mechanism itself must be validated.

1.4 Research Questions

We formulate a number of research questions aimed at addressing the problems described in Section 1.3. The central research question is as follows.

> **RQ:** *How can we ensure the correctness of software that is automatically generated from model-to-code transformations?*

This central research question is split into six more detailed research questions, from RQ_1 to RQ_6. Each of these questions is addressed in the remainder of this thesis.

Model-to-code transformations play an important part in the software engineering process for realizing the final products from models on a high level of abstraction. The main challenge of this step is to ensure that the produced code conforms to its high-level specification. As the challenge mainly originates in the gaps between DSMLs and envisaged implementation platforms, we start our research by studying the challenges and choices of implementing and verifying model-to-code transformations. To do so, we investigate challenges and choices in the context of a framework that transforms SLCO models consisting of concurrent communicating objects to parallel programs in Java. This leads to the following research question.

> **RQ_1:** *What are the challenges and choices of implementing and verifying Java code generation from concurrent state machines?*

Once challenges and choices have been identified in the context of model transformations, gaps between DSMLs and corresponding implementation platforms need to be bridged. On the one hand target programming languages may not have suitable constructs to directly implement all concepts of modeling languages. On the other hand modeling languages are often not designed to address all details needed when transforming models to concrete platforms. In search of a practical solution to this problem we implemented a framework that transforms SLCO models to Java code. The following research is related to this framework.

> **RQ_2:** *How to bridge the gaps between DSMLs and their target implementation platforms?*

To improve understandability, modifiability as well as re-usability of the implementation of models, we make a distinction between *model-generic* concepts and *model-specific* parts of models. We construct implementations of models by combining these two parts: generic code and specific code. Generic concepts of models are transformed into generic code, while aspects that are specific for concrete models are transformed into specific code. A question that naturally arises is whether we are doing the things in the right way. This in turn requires a formal logic and proof support to ensure the correctness of model-to-code transformations. One challenge regarding the correctness of model-to-code transformations is to guarantee that functional properties of the input models are preserved in their corresponding implementations. In particular, we focus on proving whether the generic code preserves certain desirable properties of models. This leads to the following research question.

> **RQ_3:** *How to show that the generic code implementing model-generic concepts preserves certain desirable properties of models?*

By constructing implementations of models for target platforms with two parts: generic code and specific code, modification of a construct of the generic code will not affect the transformation as well as other constructs of the generic code. As a consequence, each construct of the generic code can be understood in isolation which contributes to the overall understanding of the transformation. This in turn requires modular approaches for specifying and verifying each construct of the generic code. Demonstrating the ability to use tool-assisted formal verification for verifying constructs of the generic code in a modular way is therefore important. This is addressed in the following research question.

> **RQ_4:** *How to use tool-assisted formal verification to verify in a modular way a construct of the generic code implementing model-independent concepts?*

Exceptions are not considered at SLCO modeling level but have to be handled correctly in the produced code. This is because abnormal termination caused by exceptions may lead to critical issues, such as safety violations and deadlocks. In search of a practical solution to this problem, we study an exception handling mechanism called *failbox* in Java, which is intended to be applied into the framework from SLCO models to Java code. The following research question is related to this study.

> **RQ_5:** *How to ensure that generated concurrent code is robust with respect to exceptions?*

When applying the mechanism for improving the robustness of the framework from models to code, the robustness of the mechanism itself also needs to be considered. The original implementation of the mechanism only works under certain assumptions. A better implementation without assumptions is required in combination with a proof of correctness. To provide this, we present a testing approach to investigate several implementations of *failbox*. The following research is related to this testing approach.

> **RQ_6:** *How to assess that the exception mechanism failbox used in the model-to-code transformation is robust?*

1.5 Outline and Origin of Chapters

The remainder of this thesis is structured as follows. For each chapter, we indicate the research question it addresses. Also, we point out the earlier publications it is based on. All of the original work has been revised for this thesis to reflect our latest insights.

Chapter 2: Preliminaries In this chapter, we provide the preliminary concepts that are relevant to the model-to-code framework implemented and verified in this thesis. Descriptions of the domain specific Simple Language of Communicating Objects (SLCO) for modeling complex concurrent systems, Epsilon Generation Language (EGL) for transforming models to code, the formal separation logic for verifying parallel programs, together with the tool VeriFast based on the separation logic are given in this chapter.

Chapter 3: Challenges and Choices In this chapter, we address research question RQ_1 by investigating the model-to-code framework from SLCO models to Java code to derive the challenges as well as choices regarding the implementation and verification. This chapter is based on the following publication:

> [115] D. Zhang, D. Bošnački, M.G.J. van den Brand, L.J.P. Engelen, C. Huizing, R. Kuiper, and A. Wijs. Towards Verified Java Code Generation from Concurrent State Machines. *Proceedings of the Workshop on Analysis of Model Transformations (AMT@ MoDELS)*, page 64-69. CEUR, 2014.

Chapter 4: The Implementation of Model-to-Code Transformation In this chapter, we address research question RQ_2 by discussing how to bridge the gaps between domain-specific modeling languages and their envisaged implementation platforms via implementing the model-to-code framework from SLCO models to Java concurrent code. This chapter is based on the following publication:

> [115] D. Zhang, D. Bošnački, M.G.J. van den Brand, L.J.P. Engelen, C. Huizing, R. Kuiper, and A. Wijs. Towards Verified Java Code Generation from Concurrent State Machines. *Proceedings of the Workshop on Analysis of Model Transformations (AMT@ MoDELS)*, page 64-69. CEUR, 2014.

Chapter 5: Verifying Atomicity Preservation and Deadlock Freedom of Generic Code In this chapter, we address research question RQ_3 by providing a fine-grained generic mechanism to preserve the atomicity of SLCO statements in the Java implementation. The atomicity preservation and lock-deadlock freedom of generic code have been guaranteed by verifying this generic mechanism via the tool VeriFast. This chapter is based on the following publications:

> [117] D. Zhang, D. Bošnački, M.G.J. van den Brand, C. Huizing, R. Kuiper, B. Jacobs, and A. Wijs. Verification of Atomicity Preservation in Model-to-Code Transformations using Generic Java Code. *Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development*, page 578-588. SciTePress, 2016.

> [116] D. Zhang, D. Bošnački, M.G.J. van den Brand, C. Huizing, R. Kuiper, B. Jacobs, and A. Wijs. Verifying Atomicity Preservation and Deadlock Freedom of a Generic Shared Variable Mechanism Used in Model-To-Code Transformations. *Communications in Computer and Information Science*, Volume 692, page 249-273. Springer, 2017.

Chapter 6: Modular Verification of SLCO Communication Channels In this chapter, we address research question RQ_4 by verifying the implementation of SLCO communication channels in a modular way with VeriFast. To support this, a novel proof schema that supports fine grained concurrency and procedure-modularity has been proposed and integrated with VeriFast. This chapter is based on the following publication:

> [29] D. Bošnački, M.G.J. van den Brand, J. Gabriels, B. Jacobs, R. Kuiper, S. Roede, A. Wijs, and D. Zhang. Towards Modular Verification of Threaded Concurrent Executable Code Generated from DSL Models. *Formal Aspects of Component Software - 12th International Conference (FACS)*, page 141-160. Springer, 2015.

Chapter 7: Increasing Robustness via Failboxes In this chapter, we address research question RQ_5 by improving the robustness of an existing mechanism called failbox by eliminating the limiting assumptions that the original version requires. To do so, we developed several increasingly more robust implementations of failbox. This chapter is based on the following publications:

> [27] D. Bošnački, M.G.J. van den Brand, P. Denissen, C. Huizing, B. Jacobs, R. Kuiper, A. Wijs, M. Wilkowski and D. Zhang. Dependency Safety for Java: Implementing Failboxes. *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, page 15:1-15:6. ACM, 2016.

> [114] D. Zhang, D. Bošnački, M.G.J. van den Brand, P. Denissen, C. Huizing, B. Jacobs, R. Kuiper, A. Wijs, and M. Wilkowski. Dependency Safety for Java: Implementing and Testing Failboxes. *Science of Computer Programming*, Submitted. 2017.

Chapter 8: Test-Driven Evolution of Failboxes In this chapter, we address research question RQ_6. We describe a testing approach that makes it possible to develop tests for demonstrating the dependency safety weaknesses of the different failbox implementations in Chapter 7. These tests are repeatable in the sense that they give the same results for runs that may differ in scheduling, even on different platforms. This chapter is based on the following publications:

> [27] D. Bošnački, M.G.J. van den Brand, P. Denissen, C. Huizing, B. Jacobs, R. Kuiper, A. Wijs, M. Wilkowski and D. Zhang. Dependency Safety for Java: Implementing Failboxes. *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, page 15:1-15:6. ACM, 2016.

[114] D. Zhang, D. Bošnački, M.G.J. van den Brand, P. Denissen, C. Huizing, B. Jacobs, R. Kuiper, A. Wijs, and M. Wilkowski. Dependency Safety for Java: Implementing and Testing Failboxes. *Science of Computer Programming*, Submitted. 2017.

Chapter 9: Conclusions This final chapter concludes the thesis. It revisits the research questions and gives directions for future research.

1.6 Suggested Method of Reading

To reduce duplication of information, Chapter 2 provides a short description of preliminary concepts used throughout this thesis. Chapter 3 provides a detailed guideline to chapters 4 to 8. Chapter 4 focuses on the model-to-code transformation. Subsequent chapters address different aspects of the transformation, and can be read largely independently. Chapters 5 and 6 focus on feasible verifications of different aspects of the generic code of the framework. Chapters 7 and 8 focus on improving the robustness of the framework via implementing and testing of failbox constructs.

Chapter 2

Preliminaries

*In this chapter, we give a brief introduction to preliminary concepts that will be used later throughout the rest of this thesis. First, we introduce the domain-specific modeling language called Simple Language of Communicating Objects (*SLCO*) for modeling complex concurrent systems. Second, we give a short description of the Epsilon Generation Language (EGL) used for transforming* SLCO *models to Java code. Furthermore, we show the main concepts of the logic that we use to verify parallel programs, i.e., separation logic. Finally, we demonstrate essential ingredients of the verifier tool VeriFast based on separation logic for reasoning about the generated Java code from* SLCO *models.*

2.1 SLCO

The domain-specific modeling language called the Simple Language of Communicating Objects (SLCO) [11,101] was developed at the Software Engineering and Technology (TU/e) Group for specifying systems consisting of objects that operate in parallel and communicate with each other by passing signals over channels. In [41], a small system of interoperating conveyor belts is used as a case study for SLCO. The case study illustrates how different implementations for controlling this system can be generated by composing a number of transformations into different sequences of transformations.

An SLCO model consists of a number of classes, objects and communication channels. A class has state machines, variables and ports. The objects are instances of classes and their behavior is specified via state machines. The State machines of an object operate in parallel. They can access and modify variables of this object and can also send and receive signals via the ports of this object. The ports are used to connect channels to objects, and each port is connected to at most one channel. The communication between objects via channels is either bidirectional or unidirectional. SLCO supports synchronous, asynchronous lossless, and asynchronous lossy channels which are suited to transfer signals of predefined types. Each bidirectional asynchronous channel is implicitly associated with two one-place buffers and each buffer is used for one direction while a unidirectional asynchronous channel is implicitly associated with a one-place buffer.

A state machine has variables, states, and transitions. Variables of a state machine can only be accessed and modified by the state machine that contains them. States have three types: initial, ordinary and final. Each state machine has exactly one initial state, in which it starts. However, the number of ordinary and final states is not restricted, i.e., each state machine can contain any number of ordinary and final states. SLCO allows that states have multiple outgoing transitions. In such cases, if more than one outgoing transition of a state is enabled, one is taken non-deterministically from the enabled transitions. A transition from a source state to a target state is associated with a finite sequence of statements. The *simplified* SLCO models [41] only allow that each transition has at most one statement. In this thesis, we only deal with simplified SLCO models, since SLCO models consisting of transitions associated with multiple statements can always be transformed to their corresponding simplified variants via model transformations [41]. In simplified models, a transition is enabled if its associated statement is empty or enabled. Otherwise, it is blocked.

SLCO models can be created in two ways [101]. One way is to use the metamodel for SLCO which is defined using the Eclipse Modeling Framework (EMF) [1]. In this way, SLCO models are created using the standard tree-view editor provided by EMF. SLCO also has a textual concrete syntax defined via Xtext [7], which also provides users with a textual editor for creating SLCO models. Besides the textual concrete syntax, SLCO also has a graphical concrete syntax which is in the form of diagrams. The diagrams are produced by using Expand [54]; each of the diagrams is in fact a directed graph written in Dot that can be visualized with the Graphviz tool [40]. Each dot file illustrates the graphical version of an SLCO model consisting of a structure diagram, a behavior diagram and a communication diagram. The structure diagram shows the structure of classes with state machines in SLCO models; the behavior diagram represents the behavior of instances of classes (i.e., objects) which is specified by using state machines; the communication diagram demonstrates the communication between objects over channels.

In the rest of this section, we show the graphical and textual representations of an SLCO model example, respectively. The diagrams of its graphical representation are depicted in Figures 2.1, 2.2 and 2.3. In Figure 2.1, the structure diagram shows the structure of classes `P` and `Q` in our example model. Class `P` comprises three state machines, i.e., `Rec1`, `Rec2` and `SendRec`, which communicate with each other via an shared integer variable `m`. State machine `SendRec` comprises a local string variable `s` which can only be accessed by state machine `SendRec`. Class Q consists of only one state machine called `Com` that contains a local string variable `s` which is only visible inside the state machine `Com`.

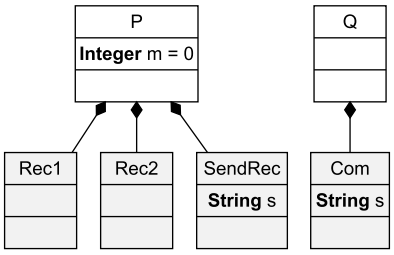

Figure 2.1: Structure diagram of an SLCO model.

The communication diagram between instances of classes `P` and `Q` in Figure 2.1 is

depicted in Figure 2.2. The object `p` is an instance of class `P`, and the object `q` is an instance of class `Q`. These two objects communicate with each other over an asynchronous, lossless channel `c1` denoted by a dashed line, lossy channel `c2` denoted by a dotted line, and a synchronous channel `c3` denoted by a solid line. The object `p` contains three ports, i.e., `In1` `In2` and `InOut`, and the object `q` also contains three ports, i.e., `Out1`, `Out2` and `InOut`. Arrowheads of lines show the directionality of channels. For instance, channel `c3` can be used to send and receive signals in both directions. Channels `c1` and `c2`, however, can only be used to send signals from object `q`. Additionally, the type of signals sent over channels `c1`, `c2`, and `c3` is restricted to `Boolean`, `Integer` or `String`.

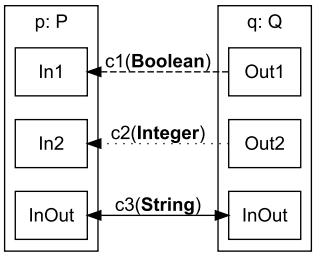

Figure 2.2: Communication diagram of an SLCO model.

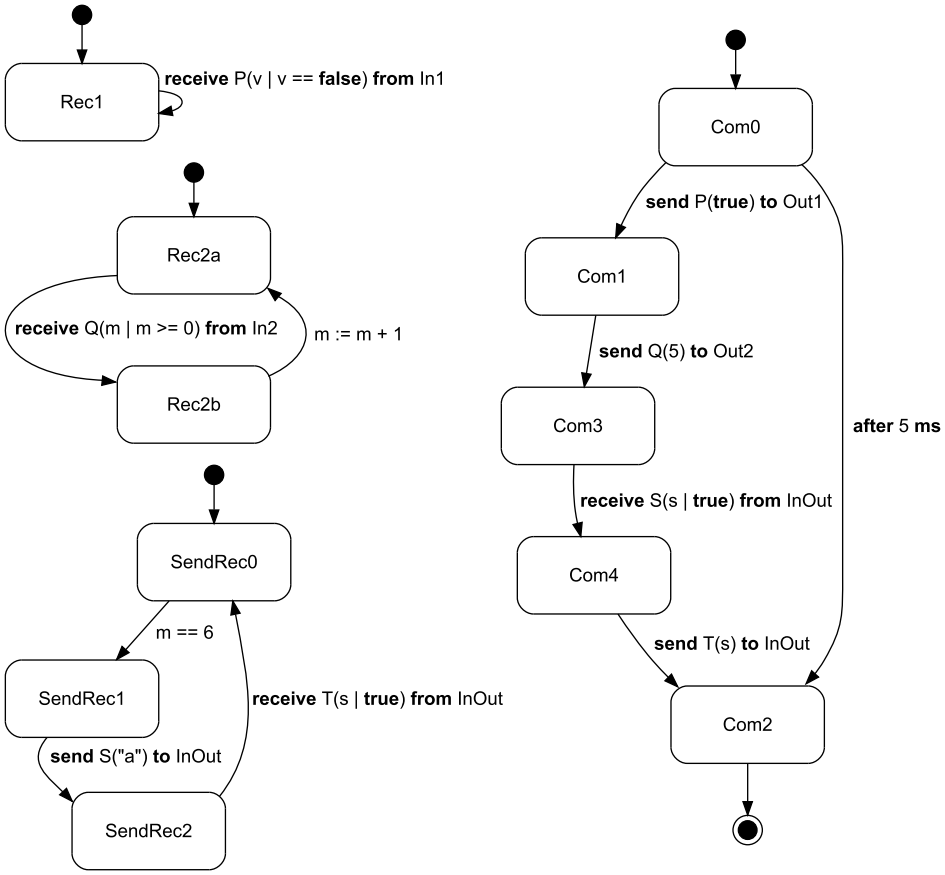

Figure 2.3: Behaviour diagram of an SLCO model.

The behavior of objects `p` and `q` is specified using state machines, which is depicted

in Figure 2.3. The behavior of object `p` is specified by state machines `Rec1`, `Rec2` and `SendRec` on the left of the figure and the behavior of object `q` is specified by state machine `Com` on the right. States `Rec1`, `Rec2a`, `SendRec0` and `Com0` are initial states and state `Com2` is a final state. State `Com0` has two outgoing transitions: one to state `Com1` and the other to `Com2`. As we only focus on the simplified SLCO models, each transition of the state machines in the figure is associated with only one statement.

SLCO supports five types of statements: SendSignal, ReceiveSignal, (Boolean) Expression, Assignment, and Delay. Each statement is either blocked or enabled and its execution is atomic. The SendSignal and ReceiveSignal statements are used to send and receive signals over channels for communication between objects. In Figure 2.3, **send** $T(s)$ **to** *InOut*, for instance, sends a signal named T with a single argument s via port *InOut*. Its counterpart **receive** $T(s|\textbf{true})$ **from** *InOut* receives a signal named T from port *InOut* and stores the value of the argument in variable s. Statements such as **receive** $P(v|v == \textbf{false})$ **from** *In1* offer a form of conditional signal reception. Only those signals whose argument is equal to **false** will be accepted and the value of the received argument is assigned to the variable v. Boolean expressions, such as $m == 6$, denote statements that block their corresponding transitions until the expressions evaluate to **true**. Time is incorporated in SLCO by means of delay statements. A delay statement blocks a transition until a specified amount of time measured in milliseconds has passed. For example, the statement **after** 5 **ms** blocks the transition from the source state `Com0` to the target state `Com2` until 5 ms have passed. Assignment statements, such as $m := m + 1$, are used to assign values to variables. Variables either belong to an object or a state machine. The variables that belong to an object are accessible by all state machines that are part of the object; the variables that belong to a state machine are accessible by that state machine.

The example model using graphical syntax, as shown in Figures 2.1, 2.2, and 2.3, can be also described using the textual syntax. The corresponding textual version of the model above is shown in Listing 2.1.

Listing 2.1: The textual SLCO model

```
1  model CoreWithTime {
2    classes
3      P {
4        variables Integer m = 0 ports In1 In2 InOut
5        state machines
6          Rec1 {
7            variables Boolean v = true initial Rec1
8            transitions Rec1toRec1 from Rec1 to Rec1 {receive
                 P(v| v == false) from In1 }
9          }
10         Rec2 {
11           initial Rec2a state Rec2b
12           transitions
13             Rec2a2Rec2b from Rec2a to Rec2b {receive Q(m| m
                  >=0)from In2}
14             Rec2b2Rec2a from Rec2b to Rec2a {m := m+1}
15         }
```

```
16          SendRec {
17            variables String s = "" initial SendRec0 state
                 SendRec1 SendRec2
18            transitions
19              SendRec02SendRec1 from SendRec0 to SendRec1 {m
                   == 6}
20              SendRec12SendRec2 from SendRec1 to SendRec2 {
                   send S("a") to InOut}
21              SendRec22SendRec0 from SendRec2 to SendRec0 {
                   receive T(s|true) from InOut}
22          }
23      }
24      Q {
25        ports Out1 Out2 InOut
26        state machines
27          Com {
28            variables String s = "" initial Com0 state Com1
                 Com3 Com4 final Com2
29            transitions
30              Com02Com1 from Com0 to Com1 {send P(true) to
                   Out1}
31              Com12Com3 from Com1 to Com3 {send Q(5) to Out2}
32              Com32Com4 from Com3 to Com4 {receive S(s|true)
                   from InOut}
33              Com42Com2 from Com4 to Com2 {send T(s) to InOut}
34              Com02Com2 from Com0 to Com2 {after 5 ms}
35          }
36      }
37    objects p: P q: Q
38    channels
39      c1(Boolean) async lossless from q.Out1 to p.In1
40      c2(Integer) async lossy from q.Out2 to p.In2
41      c3(String) sync between q.InOut and p.InOut
42  }
```

2.2 Epsilon Generation Language

The Epsilon Generation Language (EGL) [71] is a template-based model-to-text (M2T) transformation language which is provided by Epsilon. Epsilon is built on top of Eclipse and is an extensible platform of integrated and task-specific languages for model management. Due to its task-specific languages, Epsilon is well-suited to solve a range of model management problems including not only code generation but also model transformation, model comparison, merging, refactoring and validation. This facilitates the construction of the whole chain for managing SLCO models. Specifically, the transformation from SLCO models to Java code is achieved by its model-to-text transformation language, i.e., the Epsilon Generation Language.

Templates are a widely used implementation approach for code generators [14,15]. EGL also supports template-based code generation by providing a co-ordination engine that allows EGL programs to be decomposed into one or more templates. The `Template` type is the core of the engine. To simplify the creation of `Template` objects, EGL provides a built-in object called `TemplateFactory` to load templates and control the file system locations from which templates are loaded and to which text is generated.

Similar to other template-based code generators, EGL provides static and dynamic sections, from which templates may be constructed. Static sections contain code that appears verbatim in the generated text, while dynamic sections contain executable code that can be used to control the generated text. The `[% %]` tags are used to delimit dynamic sections. Any text not enclosed in such a tag pair `[% %]` is contained in a static section. Within dynamic sections, the construct `[%=expr%]` is used to append `expr` to the output generated by the transformation.

In this chapter, we only provide a brief explanation of EGL by explaining via an example how to generate code from models using it. More detailed information of EGL can be found in the Epsilon Book [71]. The example of an EGL model-to-text transformation is depicted in Listing 2.2. The transformation is used to generate a Java class called `SLCO2Java` containing the `main()` method for starting the Java program corresponding to an SLCO model. The static sections at lines 1-2 appears verbatim in the generated text, as shown at lines 1-2 in Listing 2.3. The dynamic section at line 3 uses the `allInstances` (available on every metamodel type and used to retrieve all of the instances of that metamodel type) and `first` (used to retrieve the first element of a collection) properties to find the sole instance of `Model` in the input model. The instance of `Model` is assigned to a variable named `m`. The dynamic section `[%= m.name %]` in line 4 emits the value of the `name` attribute of the `Model` instance `m`. The text that is not enclosed in the tag pair `[% %]` in line 4 appears verbatim in the generated text, as shown in line 3 in Listing 2.3.

Listing 2.2: Generating a Java class containing a main method with EGL

```
1 public class SLCO2Java {
2   public static void main(String[] args) {
3     [%var m = Model.allInstances().first();%]
4     Slco[%= m.name %] slco[%= m.name %] = new Slco[%= m.name
          %]();
5   }
6 }
```

EGL programs do not specify to which metamodel particular types belong. Instead, the user can specify the models and hence metamodels on which the EGL program operates just before executing the program. In our case, we specify the name, type, location and metamodel of SLCO when creating an Eclipse launch configuration. Executing the template in Listing 2.2, a Java class named `SLCO2Java` is generated, as shown in Listing 2.3. `CoreWithTime` is the name of an input SLCO model.

Listing 2.3: The generated Java class: SLCO2Java

```
1 public class SLCO2Java {
2   public static void main(String[] args) {
3     SlcoCoreWithTime slcoCoreWithTime = new SlcoCoreWithTime
        ();
```

```
4     }
5 }
```

2.3 Separation Logic

Concurrent programs are prone to errors because of the non-deterministic interactions between threads. Proving the correctness of a concurrent program means that this program performs its intended task for any possible scheduling of parallel actions. As a result, with standard dynamic testing techniques it is hard to ensure the correctness of concurrent programs. Static formal verification techniques are an attractive and efficient method for verifying concurrent programs, as they can formally prove that the implementation of a program satisfies its specification, i.e, any execution of the program is guaranteed to behave correctly [94]. The Owicki and Gries method [82] was invented in 1976 for reasoning about concurrent programs, building on Hoare logic [58] which is introduced for verifying sequential programs. However, this methodology breaks down when verifying parallel programs that manipulate pointers because of the possibility of race conditions involving concurrent attempts to deallocate or update a heap cell being used by another thread [30].

Separation logic [81, 86] has been developed for reasoning about shared memory which can be referenced from more than one location. It builds upon Hoare logic and in the context of concurrent programs also on the Owicki-Gries method. In this thesis, we use permission-based separation logic [24] in which the control of thread interference is established by the use of permissions of shared memory. We assume a Java-like OO programming language that features references and aliasing as well as multi-threaded concurrency. These two considerations motive us to use separation logic.

A separation logic assertion is interpreted on a program state (s, h), where s and h are a store and a heap, respectively. The store is a function mapping program variables to values, the heap is a partial map from pairs of object IDs and object fields to values. A value is either an object or a constant. To capture the heap related aspects, separation logic extends the syntax and semantics of the assertional part of Hoare logic. Separation logic adds heap operators to the usual first order syntax assertions of Hoare logic. The basic heap expressions are **emp**, i.e., the empty heap, satisfied by states having a heap with no entries, and $E \mapsto F$ (read as "E points to F"), i.e., a singleton heap, satisfied by a state with a heap consisting of only one entry at address E with content F. For instance, $o.x \mapsto v$ means that field x of object o has value v. Two heap expressions H_1 and H_2 corresponding to heaps h_1 and h_2, respectively, can be combined using the *separating conjunction operator* $*$, provided h_1 and h_2 have disjoint address domains. Expression $H_1 * H_2$ corresponds to the (disjoint) union $h_1 \uplus h_2$ of the heaps. Note that H_1 and H_2 describe two separate parts of the heap, h_1 and h_2, respectively. In contrast, the standard conjunction $p_1 \wedge p_2$, where p_1 and p_2 are separation logic formulas, corresponds to the whole heap satisfying both p_1 and p_2. Because of the domain disjointness requirement, the separation logic formula $(o.f \mapsto 10) * (o.f \mapsto 10)$ evaluates to **false**, whereas $(o.f \mapsto 10 \wedge o.f \mapsto 10)$ is equivalent to $(o.f \mapsto 10)$.

Like in Hoare logic, the triple $\{P\}\ C\ \{Q\}$, where C is a (segment of) a program and P and Q are assertions describing its pre- and post-condition, respectively, only concerns partial correctness; termination of C needs to be proven separately.

Separation logic adds to the standard rules (axioms) of the Hoare frame axioms for each of the new statements - allocation, deallocation, and assignments involving the heap cells. In some cases it is needed to embed a specification of a program segment C into a more general context. A specific axiom that allows this by enlarging the specification of a program segment C with an assertion R describing a disjoint heap segment which is not modified by any statement in C, is the *frame rule*:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

Using such called tight interpretation, separation logic assertions can describe precisely the heap footprint of a given program C, i.e., the parts of the heap which C is allowed to use. Every valid specification $\{P\}\ C\ \{Q\}$ is "tight" in the sense that every heap cell in its footprint must either be allocated by C or asserted to be active by P [33].

One of the central concepts in concurrent separation logic is *ownership*. Let l be a program component location and E a heap address. The component owns address E at location l iff E is contained in a heap corresponding to an assertion H which is true at location l. If $E \mapsto F$ is part of the heap corresponding to H, then this can be seen as an informal *permission* [24] for the verified component to read, write or dispose of the contents of the heap cell at address E. Partial permissions are introduced to allow shared ownership of variables. Ownership is split into a number of fractional permissions, each of which only allows read access. Expression $E \mapsto F$ denotes permission 1, i.e., exclusive ownership, whereas a fractional permission is expressed as $[z]E \mapsto F$ with $0 < z < 1$. Expression $[1]E \mapsto F$ is equivalent to $E \mapsto F$. Permissions can be split and merged during a proof. For instance, two fractional permissions can be merged according to the following rule: $[z]E \mapsto F * [z']E \mapsto F$, where $z + z' \leq 1$, implies $[z + z']E \mapsto F$. One acquires full ownership (and therefore write access) in case $z + z' = 1$. The split rule is analogous.

2.4 VeriFast

The VeriFast [91] tool is a program verifier for sequential and concurrent C and Java programs based on separation logic. Programs are annotated with function specifications, loop invariants, predicate definitions, and other annotations. VeriFast takes a program with its annotations and reports either that the program is memory safe, data-race free, and complies with function specifications, or it shows a symbolic execution trace that leads to a potential error.

VeriFast supports modular verification in the sense that each method is verified separately with respect to its precondition and postcondition. The precondition and postcondition of a method can be considered as a contract between callers calling the method and implementers implementing the body of the method. Callers must ensure that the precondition holds when they call the method and in return they may assume that the postcondition holds when the method returns. Implementers may assume that the precondition holds on entry to the method, and in return they are obliged to provide a method body that establishes the postcondition when the method returns [68].

VeriFast executes the method body symbolically, using a separation logic formula as the symbolic representation of memory. More concretely, VeriFast constructs a symbolic state that represents an arbitrary concrete pre-state which satisfies the precondition and checks

that the body satisfies the contract for this symbolic state [68]. The symbolic execution of a triple $\{P\}\ C\ \{Q\}$ starts in the symbolic state corresponding to the precondition P. If the triple is correct, each finite execution should eventually reach a symbolic state implying Q.

VeriFast uses fractional permission to specify the read-only sharing of chunks of memory. That is, the program memory (heap) is allowed to be broken down into separate chunks which are passed from one method to another during method calls and returns, or are distributed between concurrent threads. Each chunk in the VeriFast symbolic heap specifies a term known as its *coefficient* which represents a real number between 0 and 1, excluding 0 but including 1. A chunk with coefficient ϕ is denoted by $[\phi]o.f \mapsto v$, where $0 < \phi \leq 1$. When ϕ is 1, the coefficient is omitted and the usual $o.f \mapsto v$ is obtained. This case expresses the full ownership of the chunk, allowing exclusive write access. When ϕ is different from 1, it says that the chunk is a fraction. This case expresses the fractional permission of the chunk, allowing shared read access. These two case are in line with the concepts of separation logic explained in Section 2.3. VeriFast enforces that all fractional permissions to a memory location combined give a full write permission; the sum of the fractions should not exceed 1.

VeriFast supports custom predicates to achieve more concise contracts as well as information hiding. For instance, instead of directly referring to the internal fields of an object, preconditions and postconditions can be phrased in terms of a predicate. VeriFast also supports generics, inductive data types and fixpoint functions. Apart from these, lemma functions are introduced for proving properties about fixpoints and predicates. They allow developers to use these properties when reasoning about programs. For us, these features make it possible to verify properties of generic code of the framework that is used to generate Java code from SLCO models. More detailed information about these features is explained in the VeriFast tutorials [68, 91].

Chapter 3

Challenges and Choices

A question that naturally arises for model-to-code transformations is how to guarantee that functional properties of the input models are preserved in the generated code. We consider this issue for a framework that implements the transformation from the modeling language SLCO to Java code. Our aim is to specify generic modeling constructs and verify the implementation of those constructs in a modular way. We identify several challenges in the implementation and verification of the transformation. This chapter serves as an introduction for Chapters 4 to 8.

3.1 Introduction

One of the main goals of MDSE is to produce source code automatically from models. This goal can be achieved by providing model-to-code transformations that ensure the conformance of the produced code to its high-level specification.

The issues presented mainly originate in the lack of correspondence between model-oriented primitives in domain-specific modeling languages and their counterparts in programming languages. Implementing the semantics of high-level modeling languages is a challenging task because a target programming language may not have suitable constructs to directly implement all concepts of a modeling language (e.g., atomicity, non-determinism). Another challenge is that high-level modeling languages are often not expressive enough to address all details needed when generating executable code from them (e.g., exceptions, fairness).

Model-to-code transformations as well as code produced by them may not be free of bugs and therefore must be verified and validated. Hence, once we figure out how to solve the above mentioned discrepancy issues, we need to show that we are doing the things in the right way. This in turn requires a formal logic and proof support to ensure the correctness of produced code and transformations themselves. Besides that, quality aspects, like modularity and efficiency, need to be considered, which impose additional challenges. Of course, all these challenges are closely intertwined. For example, the correctness and efficiency aspects can influence the choices between solution alternatives, i.e., how the

particular concepts are implemented.

In this chapter, we investigate challenges regarding model-to-code transformations, in the context of a framework that transforms SLCO models consisting of concurrent, communicating objects to parallel programs in Java.

At the SLCO model level, the execution of each SLCO statement is atomic. This facilitates reasoning about the behavior of concurrent systems at an abstract level. However, at the Java level, the level of granularity of the execution of a statement is more fine-grained than the one in SLCO, i.e., each SLCO statement is mapped to a block of instructions in Java. The concurrent execution of these instructions results in better performance. Thus, the challenge is how to transform an SLCO statement to a block of Java instructions with equivalent observable behavior but where the strong atomicity notion is replaced with a weaker one in Java, maintaining better efficiency.

Modeling languages usually feature non-determinism whereas the programming languages are essentially deterministic. Hence, implementing non-determinism is a standard issue in model-to-code transformations. SLCO too allows a state to have multiple outgoing transitions and the choice between enabled transitions is non-deterministic. In particular, one needs to resolve the cases when one of the transitions of the non-deterministic choice is blocked. Mimicking (a combination) of the blocking state of a transition and non-deterministic choice is possible in Java. However, not all of the solutions work efficiently. As efficiency is an important requirement for concurrent programs, the challenge is to choose an efficient implementation while ensuring that the semantics of blocking state and non-determinism before and after transformation are the same.

In SLCO the issue with the blocked choice is more complicated in the presence of communication. State machines in different SLCO objects communicate with each other over synchronous and asynchronous channels supporting conditional signal reception statements. Such channel operations - and actually the channels themselves - have no direct counterparts in Java that match their semantics. They are difficult to implement, since state machines that are executed in parallel do not have the global overview of the behavior of the SLCO model consisting of these state machines. Thus, a complex protocol has to be applied in the implementation to ensure that the behavior of channels before and after transformation is the same.

Modularity is one relevant quality attribute of model transformation [103]. By systematically separating and structuring a model transformation into small separated and isolated modules, the understandability, modifiability and re-usability of a model transformation are improved as well. For this purpose, we define the transformation framework from SLCO models to Java code in a modular way. Luckily, most model transformation languages have support for structuring model transformations by packaging transformation rules into modules [34]. However, modularity is not limited to model transformations. The code transformed from models should be divided into modules as well. The challenge is how to construct small and separated modules that are well isolated instead of one monolithic chunk of tightly-coupled code in the implementation. To this end, we make a distinction between *model-generic* parts and *model-specific* parts of SLCO models. Model-generic parts of an SLCO model are transformed into generic modules in Java while aspects that are specific for an individual SLCO model are transformed into specific Java code. The specific code is combined with the generic Java code to obtain complete, executable code that should behave as the SLCO model specifies.

An exception is an issue that arises during the execution of a program, which is not

considered at the SLCO model level. However, abnormal terminations caused by exceptions may lead to critical issues for concurrent programs, such as safety violations (caused by inconsistent data structures) and deadlocks, which is not desired in the implementation of an SLCO model. The problem of existing exceptions handling mechanisms in Java is that they do not handle abnormal termination in a compositional manner. More specifically, the failure of a code block should not necessarily terminate the entire application. For this purpose, we considered applying an existing mechanism called failbox [66] to the implementation. Unfortunately, the original implementation of failbox is not implemented in Java and only works under some assumptions. Thus, the challenge is to improve it by eliminating these assumptions and also test the robustness of the improved version.

Fairness is another important feature of multi-threaded programs. In the absence of fairness guarantees, some threads may make no progress when multiple threads are runnable. The reason is that certain threads may continuously grab the CPU time. As SLCO was introduced to model complex concurrent systems by means of state machines and each state machine is mapped to an individual thread in our framework, the concurrency among multiple threads may raise fairness issues as well. The problem is that at the SLCO modeling level fairness is not considered, in line with the high-level abstraction. At the Java level not all constructs in Java library code support fairness policy as well. Therefore, the challenge when transforming SLCO models to good implementations is how to ensure the fairness between threads in the implementation to some extent by using the right constructs obtained from Java library code.

The rest of the chapter is organized as follows. In Section 3.2 challenges on transforming an SLCO statement to a block of Java instructions with equivalent observable behavior but where the strong atomicity notion is replaced with a weaker one in Java are provided. Mechanisms in Java to mimic the blocking state of a transition and non-deterministic choice between multiple transitions in SLCO are discussed in Section 3.3. In Section 3.4 we present challenges on how to implement SLCO channels in Java. The challenge caused by exceptions in the implementation and its solutions are discussed in Section 3.5. In Sections 3.6 and 3.7 issues and their solutions regarding modularity and fairness are described, respectively. Section 3.8 summarizes this chapter.

3.2 Atomicity

Atomicity is an important concept in SLCO models for concurrent executions of state machines. SLCO provides a strong notion of atomicity – each statement is atomic. Semantically, it means each statement s must execute as though there is no interleaved computation, i.e., no other state machines are running. This strong notion makes reasoning about the behavior of concurrent state machines easy.

Since the level of granularity of the execution of a statement in Java is more fine-grained than the one in SLCO, not only the strong notion but also a weaker notion of atomicity can be defined for a statement in Java. The weaker notion allows interleavings between Java instructions originating from different SLCO statements which provides a better performance of multi-threaded programs. Therefore, to present a well-performing, efficient implementation of SLCO models, one of the important choices is where to set the granularity of atomicity in implementations.

Specifically, each statement s in a state machine M can be transformed into a block of

Java instructions $\sigma = \mathtt{s}_0; \mathtt{s}_1; \ldots; \mathtt{s}_n$ which are executed by a thread t_M. The atomicity of s in an SLCO model can be formulated as no instruction $\mathtt{s}'$ of any thread $t \neq t_M$ is allowed to be executed between the beginning and end of the execution of σ in Java. This can be achieved via coarse-grained locking mechanisms. For instance, the access to all shared variables in a Java application can be controlled by a single lock. However, when taking this approach, the problem is that it excludes the possibility for threads to run truly concurrently, e.g., in cases where they access different shared variables, and therefore do not interfere with each other. Hence, implementing atomicity in this strict sense is undesirable for performance reasons.

To achieve true parallelism, the notion of strong atomicity is usually replaced with weaker notions that still ensure non-interference. The weaker notions allow instructions of different blocks to be executed concurrently as long as the instructions do not interfere with each other. One such version, sometimes called *serializability* [21] guarantees that for any concurrent execution of (atomic) Java blocks there exists a sequential execution of those blocks that is indistinguishable from the concurrent execution, in terms of the final effect on the global system state.

In order to realize this kind of weak notion of atomicity in Java in a setting where multiple threads may access the same shared variables simultaneously, it must be ensured that an instruction $\mathbf{s}$ of some thread t cannot affect the variables accessed by the instructions in a block σ of thread $t \neq t_M$ running concurrently. This can be achieved by a fine-grained locking mechanism – lock each shared variable with an individual lock. Such kind of fine-grained locking mechanism improves the performance of programs, but also poses problems. For instance, a deadlock may occur if circular waiting is possible when for a set of threads each thread waits for another one to release a lock. Moreover, several locks are often required before executing a single statement. Omitting one of them potentially leads to data races, i.e., unsynchronized conflicting accesses of the same variable by multiple threads. Consequently, the challenge of applying the fine-grained locking is how to avoid deadlocks and data races. The deadlocks are addressed by using the technique of ordered locking when requiring locks, which is presented in Chapter 5. The ordered locking guarantees that when multiple threads compete over a set of variables, one thread is always able to acquire access to all of them. In this way, deadlocks can be prevented. Additionally, omitting locks can be avoided by automatically creating a list datatype and then storing all shared variables involved in a single statement to this list during the model-to-code transformation. Consequently, the lock of each element in the list can be acquired sequentially before the execution of a statement.

Reasoning about the preservation of atomicity of blocks in programming languages is another challenge. The verification methodology should be able to verify that the implementation satisfies its specification in terms of non-interference. On top of this, the absence of data races and deadlocks of the implementation should also be verified. To address this, we formally prove the implementation against a specification of non-interference using the VeriFast tool [6]. This tool is also suitable to deal with race conditions and deadlocks using the concept of ownership of shared resources for multi-threaded, object-based programs. The implementation of atomicity as well as its verification are addressed in Chapter 5.

3.3 Transitions

As mentioned in Section 2.1, in simplified SLCO each transition is associated with at most one statement. A transition is enabled if its statement is empty or its statement is enabled. Otherwise, it is blocked. Moreover, if there are multiple enabled outgoing transitions from a state, then the choice among them is nondeterministic. Thus, two challenging issues have to be considered in terms of the implementation as well as their verification: the blocking state of a transition and non-deterministic choice between transitions.

3.3.1 Non-deterministic Transitions

Non-deterministic choice between transitions is another challenging issue when implementing SLCO models at lower level languages. One simple solution is to implement SLCO models without preserving non-determinism in their corresponding Java code. Specifically, only one transition from multiple outgoing transitions of a state is chosen and then preserved during the model-to-code transformation. This approach can be adopted when the goal is to see one of the possible implementations of a model, rather than a full coverage of all options like in model verification. This approach is not taken into account in our research.

To preserve the non-determinism feature in the Java implementation, one possibility is to list all outgoing transitions with a fixed order in a Java blocking code block. According to the order, each of them is checked one by one within a loop until one of them is enabled and then executed. However, the problem is that each transition is associated with a potential priority that corresponds to its order in the block. As a consequence, transitions with lower priority may never be executed when transitions with higher priority are enabled. This case may lead to fairness issues which are undesirable in the implementation.

Another possibility is to make the choice between outgoing transitions from a state in a fair way. This can be achieved by applying a pseudo-random number generator in Java. The input of the pseudo-random generator can be limited to only enabled transitions. In this case, all outgoing transitions from a state are evaluated in advance and then one of the enabled transitions is selected using the pseudo-random generator. However, the efficiency of this solution is quite low in cases where most of the outgoing transitions of a state are enabled. Additionally, we use fine-grained locking for an atomic SLCO statement, i.e., each shared variable is associated with a single lock. Hence, locks involved in statements corresponding to all outgoing transitions of a state need to be acquired before one of enabled transitions is taken, which also decreases the efficiency. Alternatively, a pseudo-random number generator can select one of outgoing transitions randomly without distinguishing between enabled and disabled transitions. In this case, one of the outgoing transitions is chosen randomly and then evaluated. If it is enabled, this transition is taken. Otherwise, the pseudo-random generator continues the selection process. This approach is more efficient, since the enabled transition can be executed immediately after it is chosen by the pseudo-random number generator. We apply this approach to our current implementation, as shown in the generated code on state machines in Chapter 4. The verification of non-determinism, another challenge, is not presented in this thesis but is considered as future work.

3.3.2 Blocking Transitions

The blocking state of a transition in the implementation can only be assessed by initiating its associated statement and then observing whether it blocks. Each blocking transition with its statement in SLCO can be modeled by means of guarded blocks in Java. The guarded blocks in Java can be realized using two mechanisms: a busy-waiting loop and a wait-notify mechanism.

A busy-waiting loop means that a thread continuously checks whether the condition of the guarded block evaluates to true. However, the continuous checking may waste CPU cycles, which is inefficient. Additionally, a low priority thread may never get a chance to execute when a high priority thread goes into a busy-waiting loop.

A more efficient solution is to use a wait-notify mechanism that enables threads to become inactive while waiting for signals. Java provides interfaces and classes to implement the wait-notify mechanism. For instance, the class `java.lang.Object` defines three methods, i.e., `wait`, `notify`, and `notifyAll`, to facilitate this. A thread calls the `wait` method on an object to release the acquired monitor on this object and leaves the processor to be used by other threads. The thread becomes inactive and waits until another one notifies threads waiting on this object's monitor to wake up either through a call to `notify` or `notifyAll`. Note that these three methods must be called in methods or blocks marked with the `synchronized` keyword which requires the locking to obey a built-in acquire-release protocol, i.e., a lock is acquired on entry to a `synchronized` method or block, and released on exit. However, when one thread releases a lock and another thread may acquire it, there is no guarantee about which of any blocked threads will acquire the lock next or when they will do so [75]. In particular, there are no fairness guarantees. Besides such monitor methods, the interface `java.util.concurrent.locks.Condition` provides a framework for locking and waiting for conditions, where a lock replaces the use of `synchronized` methods and blocks and a condition replaces the use of the `Object` monitor methods. `Condition` objects provide a means for a thread with the ability to suspend its execution until the condition is true. A `Condition` object is necessarily bound to a `Lock` and can be obtained using the `newCondition` method. Locks in Java associated with conditions support an optional fairness policy, i.e., under contention locks favor granting access to the longest-waiting thread. This avoids starvation of threads. Thus, this solution can be considered when taking the fairness property into account for implementing SLCO models.

However, the wait-notify mechanism also has some issues. One issue is spurious wakeups [2], i.e., threads may wake up even if methods `notify` or `notifyAll` have not been called. Another issue is that a waiting thread may wait forever. This happens when there are two threads where one calls method `notify` on a object before the other calls method `wait` on the same object.

To apply the wait-notify mechanism in the implementation, the two issues above need to be addressed. The first issue is fixed via nesting each call of the method `wait` inside a while loop with a conditional expression. If a blocked thread wakes up without receiving a signal, the while loop will execute once more, causing this awakened thread to go back to wait. The second issue is addressed by adding a timeout time to a `wait` call. The method `wait` with timeout information causes the current thread to wait until either another thread invokes the `notify` method or the `notifyAll` method on the same object, or a specified amount of time has elapsed. In this way, a blocked thread can always wake up

and re-check the condition for the while loop when the signal to wake up is missed.

The implementation of the blocking state of a transition using the wait-notify mechanism is reflected in the implementation of SLCO statements in Chapter 4, as the blocking state of a transition depends on the state of its associated statement. Currently, the tool VeriFast does not support the verification of methods `wait`, `notify` and `notifyAll` yet. An interesting direction for future work is to define these methods' specifications in VeriFast and then verify the wait-notify mechanism which is applied in our framework. Alternatively, we can also explore other verification tools with support for the Java wait-notify mechanism.

3.4 Channels

As discussed in Section 2.1, state machines in different objects in SLCO communicate with each other by sending signals through asynchronous and synchronous channels. The channels can be unidirectional or bidirectional, which allows communication in one or two directions. As a bidirectional channel can be easily replaced with two undirectional channels via model-to-model transformations, we only focus on the implementation of unidirectional channels in this thesis.

In particular, SLCO allows for communication with conditional reception over channels. Conditional reception states that a state machine, as a receiver, only accepts a signal if it has appropriate arguments and satisfies conditional expressions over the arguments before communicating with another state machine. Such channel operations - and actually the channels themselves - have no direct counterparts in Java that match their semantics. The implementation of conditional reception is more complicated in the presence of synchronous communication, as state machines that are executed in parallel do not have the global overview of the behavior of the SLCO model consisting of these state machines. Thus, a complex protocol has to be applied in the implementation to ensure that the behavior of channels before and after transformation is the same.

3.4.1 Asynchronous Channels

Each unidirectional asynchronous channel in SLCO is implicitly associated with a one-place buffer. If the buffer corresponding to a channel is not empty, statements that send signals over this buffer block. Otherwise, such a statement is enabled, and adds a signal to the buffer. Similarly, if the buffer corresponding to a channel is empty, the signal reception statements that receive signals via this channel are blocked. If the buffer contains a signal but it does not have appropriate arguments or conditional expressions over the arguments are false, the signal reception statements are also blocked.

As a first approximation to implement SLCO asynchronous channels in Java, we use a class `ArrayList` instance with capacity one as the implicit one-place buffer that is associated with an asynchronous channel. If SLCO is ever extended to support more than one-place buffers for each asynchronous channel, the buffer implemented using `ArrayList` can be easily adjusted to the one with more than one-place. Additionally, `ArrayList` appends the specified element to the end of this list via method `add` and removes the element at a specified position in this list via method `remove`. These two methods can be used as sending and receiving operations of SLCO channels, respectively. The accesses of sending and receiving operations between threads are controlled by using the Java class

`Semaphore`.

This implementation can be considered as a simplified version of SLCO's asynchronous channels, since it does not cover the conditional reception feature. Also, the channel implementation itself does not support blocking sending and receiving operations that wait for the corresponding buffer to become non-empty when receiving a message, and wait for space to become available in the buffer when sending a message. The advantage of this simplification facilitates the verification of the channel implementation itself.

The implementation of asynchronous channels as mentioned above as well as its verification is presented in Chapter 6. One of the aims in this thesis is to show that modular verification of model-to-code transformations of multi-component models is necessary and feasible. As a first step, we focus on how to formally specify the behavior of a model independent concept, i.e., the channel, using the VeriFast tool introduced in Chapter 2, such that modular verification of code is possible. Additionally, we introduce a novel module specification schema which improves the modularity of the VeriFast approach.

To support the above mentioned features of asynchronous channels, we then develop an alternative version incorporating more advanced concurrency features of Java. The package `java.util.concurrent` contains a set of classes that makes it easier to develop concurrent (multi-threaded) applications in Java. For example, the `BlockingQueue` in the package `java.util.concurrent` is a queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element. Accordingly, blocking features can be added to the former implementation by replacing `ArrayList` with `BlockingQueue`. Another important feature of `BlockingQueue` is that all queuing methods achieve their effects atomically, using internal locks or other forms of concurrency control, i.e., all operations on `BlockingQueue` are thread-safe. As a result, explicit locking to control accesses on a shared `BlockingQueue` between threads can be avoided. Furthermore, a conditional reception feature is also added to the implementation. This is achieved via the method `peek`, which retrieves but does not remove the element from the queue. Before taking an element from the queue, a thread needs to retrieve it and check its content. In this way, the element can only be taken via the method `take` when the element is expected.

The improved implementation of asynchronous channels is presented in Chapter 4. To be able to verify the improved implementation in future work, we still need to extend the tool VeriFast with specifications of operations of class `BlockingQueue`. The specifications involve how to specify built-in locks of the operations and how to specify the blocking state of each operation.

3.4.2 Synchronous Channels

Synchronous communication is a typical example of a construct at a high level of abstraction that is often present in formal modeling languages, but does not always have a directly corresponding concept in the target programming language. One possibility is to describe the behavior of synchronous communication in terms of communication over an asynchronous channel at the modeling level. This can be achieved by a model-to-model transformation which transforms an SLCO model to a different SLCO model with equivalent observable behavior but where synchronous communication is replaced with asynchronous communication [41]. In this thesis, however, we focus on how to directly implement the

construct of SLCO's synchronous channel in the Java programming language, as it is desirable to preserve modeling constructs as directly as possible at the implementation level.

SLCO supports synchronous communication with conditional reception. Specifically, a state machine, as a receiver, is ready only when a signal sent over the synchronous channel is expected by the receiver. The signal is expected when it has appropriate arguments and conditional expressions over these arguments are true for the receiver. If a state machine, as a sender, sends a signal over a synchronous channel before its corresponding receiver reaches the point to retrieve the signal, the sender waits until the receiver retrieves the signal. Similarly, if the receiver reaches the point of receiving a signal before the sender sends one, the receiver waits until an expected signal becomes available. SLCO also supports synchronous communication in cases where SLCO models contain states with multiple outgoing transitions and at least one of these transitions is associated with a `SendSignal` statement or a `ReceiveSignal` statement. In such cases, if only one side, a sender or a receiver, reaches the point of sending or receiving a signal, it will not wait until the other side is ready and it will choose one of the other enabled outgoing transitions to execute.

Java supports synchronous communication via class `SynchronousQueue` in the package `java.util.concurrent`. That is, each insert operation of the queue must wait for a corresponding remove operation by another thread, and vice versa. Synchronous queues are similar to rendezvous channels used in CSP [59] and Ada [62]. However, this kind of synchronous communication is different from the synchronous communication in SLCO. One difference is that at the SLCO level state machines for receiving signals over synchronous channels have to check whether the content of signals are as expected before starting synchronous communication with other state machines. However, at the Java level threads do not need to know the content of elements before taking them from synchronous queues. Another difference is that the execution of a state machine in SLCO may not be blocked at a send operation of the synchronous channel when no corresponding receive operation is executed by another state machine. This is because a send operation of the synchronous channel in SLCO may be associated with a non-deterministic transition. In this case, the state machine is able to check other outgoing transitions from the same state and select an enabled one to execute. However, at the Java level a thread is blocked at an insert operation of the synchronous queue when there is no corresponding take operation executed by another thread. In order to bridge these gaps between synchronous communication in Java and in SLCO, an additional protocol has to be introduced.

The first gap can be bridged by using a `SynchronousQueue` together with a `BlockingQueue` in Java to implement the synchronous channel in SLCO. Specifically, the `SynchronousQueue` is used to synchronize the synchronous communication between sending and receiving threads when both of them are ready to communicate. The `BlockingQueue` is used to achieve conditional receiving operations of SLCO synchronous channel. More precisely, the sending thread first puts a message in the `BlockingQueue`, after which the sending thread inserts the same message in the `SynchronousQueue` via its `put` method. After these two steps, the sending thread will be blocked by the `put` operation of the `SynchronousQueue` if its corresponding receiving thread does not reach the point to take the message from the `SynchronousQueue` via its method `take`. In this way, a receiving thread first checks the content of the message in the `BlockingQueue` and then decides whether to start the synchronous communication with

the sending thread over `SynchronousQueue`.

Based on the above solution for the first gap, the second gap mentioned can be filled by adding timeout information to insert and take operations of `SynchronousQueue` when implementing SLCO's synchronous channels in Java. A thread inserts a message into the `SynchronousQueue` and waits, if necessary, up to the specified wait time, for another thread to take it. In this way, after the specified wait time, the inserting thread is able to exit the blocking state when there is no corresponding take operation executed by another thread. This solution is discussed in more detail in Chapter 4. As a next step of our research, specifications of synchronous communications via methods `take` and `put` need to be defined in VeriFast.

3.5 Robustness

An exception is an issue that arises during the execution of a program, which is not considered at the SLCO modeling level. When an exception occurs, the normal flow of the program is disrupted and the program terminates abnormally. Abnormal terminations may lead to critical issues, such as safety violations (caused by inconsistent data structures) or deadlocks, which is not desired. Therefore, towards the constructing of robust and reliable generated code from models, one of the challenges is to ensure that when an operation in generated code fails, code that depends on the operation's successful completion is not executed.

Exception handling is an important aspect of writing robust Java applications. When an error occurs in a Java program, it usually results in an exception being thrown. Exceptions that arise during program executions can be handled via the basic try-catch mechanism in Java. A try block contains a block of program statements within which an exception might occur. A try block is always followed by a catch block, which handles the exception that occurs in the associated try block. The corresponding catch block starts executing after an exception of a particular type has occurred within the try block.

However, the basic try-catch mechanism in Java is not sufficient to ensure that when an operation fails caused by exceptions, code that depends on the operation's successful completion is not executed. To address this kind of issue, a language extension called *failbox* was proposed in [66]. Failbox ensures that if an operation fails, no code that depends on the successful completion of the operation is executed anymore nor will wait for the completion. Therefore, we can apply the failbox mechanism instead of default try-catch mechanism to improve the robustness of the implementation.

The failbox mechanism allows dealing with exceptions compositionally. However, the problem is that the original failbox implementation requires the assumption of absence of asynchronous exceptions inside the faibox code. This implementation is in Scala [4] and there is no corresponding Java implementation. To this end, we implement the failbox in Java and also improve it by eliminating the assumption required in the original failboxes implementation. This assumption is eliminated in an incremental manner, through several increasingly more robust implementations. For each implementation we analyze the vulnerabilities and argue the remedies in the next implementation. The incremental implementations are presented in Chapter 7. Additionally, we present a testing approach to investigate whether the vulnerabilities of each implementation of failbox are realistic and the remedies proposed in the next implementation are effective. This testing approach

enables us to generate asynchronous exceptions in a controlled manner for concurrent programs. The tests are repeatable in that they give the same results for runs that may differ in scheduling, even on different platforms. More details of this testing approach can be found in Chapter 8.

Verification of exception cases can also be challenging. In such cases, the verification technique will require the specification taking exceptions into account. However, these specifications are often not available in practice.

3.6 Modularity

We are aiming to automatically generate Java code from SLCO models in a modular way, where generic features of models can be separated into independent modules. This allows the implementation of each module to be updated without affecting other modules. In this way, the maintainability and re-usability of the implementation can be improved. Additionally, each module can be analyzed separately, allowing for modular verification.

To achieve modular implementation and verification, we divide the transformation into two parts as described in Chapter 4, one part transforming SLCO concepts into generic code, and the other part transforming the aspects that are specific for the particular input SLCO model into specific code. An example of a generic SLCO concept is the communication channel, while a particular state machine is an example of a concept specific for a given SLCO model. In this way, the implementation of generic concepts can be updated without affecting the overall transformation machinery. Additionally, generic code only needs to be verified once. Each class of the generic code can be specified and verified in isolation, allowing for modular verification. In Chapter 5 we discuss how we implement, specify, and verify a generic protection mechanism to access shared variables involved in atomic SLCO statements in a modular way. The solution and its specification and verification can be regarded as a reusable module to safely implement atomic operations in concurrent systems. In Chapter 6 we investigate the modular specification and verification of another example of model-generic parts of SLCO models, i.e., the communication channel which is generic in the sense that it is reused in the translation of all specific SLCO models consisting of objects that communicate with each other through channels.

Besides the verification of generic code, specific code should also be verified. This can be addressed by generating annotations along with the code from transformations. This complementary approach allows fully automated program proofs from the model-to-code generator. This part of work can be considered as a valuable direction for future work, as the transformation framework can then be fully verified and thus its correctness can be ensured. However, this step remains challenging to implement and maintain because the annotations are cross-cutting concerns, both on the object-level (i.e., in the generated Java code) and on the meta-level (i.e., in the generator).

3.7 Fairness

SLCO was introduced to model complex concurrent systems by means of state machines in combination with shared variables and channels. At the SLCO model level, no fairness is assumed. However, when implementing SLCO models for target platforms, each state machine can be mapped to an individual thread. As a consequence, the concurrency

among multiple threads raises fairness issues. For instance, in the absence of fairness guarantees in a concurrent program, a given thread may starve because other threads may continuously grab the CPU time. The thread is then starved to death, which is not the intended meaning of the corresponding SLCO model. As a result, if the quality of the implementation of SLCO models depends upon a fairness property, this must be taken into account for the implementation in some way, especially when target platforms support fair scheduling or constructs with fairness policy.

At the Java level, some constructs in Java library code supporting fairness policy can be used in the implementation. An example of these constructs in Java is class `ReentrantLock`. Its constructor accepts an optional fairness parameter. Locks favor granting access under contention to the longest-waiting thread when this parameter is set to true. This class is used to implement a fine-grained locking mechanism for protecting shared variables in our framework, as shown in Chapter 4. Additionally, `ArrayBlockingQueue` also supports an optional fairness policy for ordering waiting threads in FIFO (first-in-first-out) order in the buffer. In Chapter 4 we use this class to implement the implicit buffer associated with SLCO channels to ensure fairness between waiting threads for accessing a channel. Furthermore, as discussed in Section 3.3.2, the blocking state of a transition can be implemented via the wait-notify mechanism using either `java.lang.Object` or `java.util.concurrent.locks.Condition`. The methods `wait`, `notify` and `notifyAll` of `java.lang.Object` must be called in Java `synchronized` blocks which do not guarantee fairness. Consequently, there are no guarantees about which of the threads in a wait set will be chosen in a `notify` operation, or which thread will grab the lock first and be able to proceed in a `notifyAll` operation [75]. However, locks associated with conditions support an optional fairness policy, i.e., under contention locks favor granting access to the longest-waiting thread. This in turn avoids starvation of threads. For this purpose, the blocking state of a transition is implemented using conditions, which is presented in Chapter 4 as well.

Therefore, the current Java implementation of SLCO models in our framework ensures fairness properties in the sense that we rely on the constructs supporting fairness provided by the Java library.

3.8 Conclusions

In this chapter, we identified several challenges in the context of Java code generation from SLCO models. Exploring possibilities to address each challenge is time-consuming and challenging. This is because the Java programming language does often not have suitable constructs to directly implement all concepts of SLCO. This in turn requires that developers have to investigate more time to explore existing Java constructs and then find a way to mimic each SLCO construct with equivalent observable behavior by using related existing Java constructs. Moreover, all implementation details which are not defined at the abstract model level need to be figured out by developers when implementing models at the low programming level. To have good implementations, quality aspects, such as modularity, efficiency, robustness and fairness, need to considered as well. On the other hand, this consideration imposes additional challenges, as quality aspects are often intertwined. The choice between solution alternatives may be influenced when considering different quality aspects.

To be able to transform an atomic SLCO statement to a block of Java instructions with equivalent observable behavior, one possibility is to use coarse-grained locking mechanisms. This solution makes reasoning about the behavior of concurrent programs easy. However, it is undesirable for performance reasons regarding true parallelism of concurrent programs. To gain better performance, we proposed another possibility by using fine-grained locking mechanisms to ensure the equivalent observable behavior between an SLCO statement and its corresponding block of Java instructions. We also discussed the way how to avoid deadlocks often occurring in fine-grained locking mechanisms, i.e., locks are acquired in a fixed order.

Another challenge is that one needs to resolve the cases when at least one of the transitions in a non-deterministic choice is blocked in SLCO. To mimic the combination of the blocking state of a transition and non-deterministic choice, we first explored solutions to deal with non-determinism. As the execution of a program in Java is deterministic, one possibility is to rule out the non-deterministic feature when implementing SLCO models. Apart from ruling out the non-determinism, the possibilities to mimic the non-deterministic choice by using a pseudo-random number generator in Java are also discussed. In terms of mimicking the blocking state of a transition, several possibilities via using a busy-waiting or a wait-notify mechanisms are proposed. Using the wait-notify mechanism is more efficient, as it allows threads to become inactive while waiting for signals. However, the wait-notify mechanism in Java has two issues, namely spurious wakeups and missing signals. These two issues have to be fixed when applying the wait-notify mechanism to the framework. We also presented solutions to address these issues.

In SLCO the issue with the blocked choice is more complicated in presence of communication over synchronous and asynchronous channels between state machines. To address this issue in terms of implementing SLCO channels, we explored possibilities to implement SLCO channels by investigating Java constructs supporting blocking operations and synchronous communications in the package `java.util.concurrent`, such as `BlockingQueue` and `SynchronousQueue`.

Modularity as one relevant quality attribute of model transformations needs to be considered when implementing model transformations. Each module can be analyzed separately, allowing for modular verification. As a result, the verification of the framework from SLCO models to Java code can be set up in an incremental way instead of verifying the whole framework in one step. For this purpose, we proposed to define the model transformation itself in a modular way by packaging transformation rules into modules regarding different SLCO constructs. Also, we proposed to make a distinction between model-generic parts and model-specific parts of SLCO models, which are transformed into generic Java code and specific Java code, respectively.

Issues caused by exceptions and lacking of fairness are not addressed in an SLCO model, since high-level abstract modeling languages are not often expressive enough to address all details needed when generating executable codes from them. The challenge when transforming SLCO models to good implementations is that exceptions and fairness have to be considered in some way. To this end, an existing mechanism called failbox is investigated to handle the issues caused by exceptions. Moreover, by using the appropriate constructs obtained from Java library code, we discussed possibilities to ensure the fairness between state machines when they access shared resources.

Based on choices we made in this chapter, we present our concrete solutions to address challenges regarding discrepancy issues between SLCO and Java in Chapter 4. Once we

figure out how to bridge the gap between SLCO and Java, we need to show that we are doing the things in the right way. To this end, in Chapters 5 and 6 we demonstrate how to specify and verify the correctness of the corresponding Java implementations of two examples of SLCO constructs. In Chapters 7 and 8, we focus on how to deal with issues caused by exceptions via investigating and improving an existing exception handling mechanism, i.e, failbox, which plays a fundamental role for the further research on improving the robustness of the implementations of SLCO models.

Chapter 4

The Implementation of the Model-to-Code Transformation

Model-Driven Software Engineering (MDSE) aims at improving the productivity and maintainability of software by raising the level of abstraction at which software is developed. Starting with an abstract model specified in a domain-specific modeling language (DSML), more detailed information is added to it in an incremental manner through a sequence of model-to-model transformations, until the model can be transformed to source code. The last transformation from model to source code, i.e., model-to-text transformation, can be either manual or automatic. However, the manual transformation is slow and error-prone, while the automatic transformation helps to increase effectiveness of complex software production by reducing the cost and time associated with the coding effort. Therefore, in this chapter, we present an automatic model-to-code transformation from SLCO *models to concurrent Java code using the Epsilon Generation Language (EGL). The challenges discussed in Chapter 3 regarding implementation are also addressed in this chapter.*

4.1 Introduction

The transformations from models to their corresponding source code can be performed in a manual way. However, such manual transformations present a large work load. Additionally, manual transformations are slow and error-prone. For these reasons, we propose an automatic model-to-code transformation from SLCO models to concurrent Java code, which not only reduces the cost and time associated with the manual coding effort but also eliminates the introduction of human errors in the transformation process.

In Chapter 3, we investigate a number of challenges to implement the transformation from SLCO models to Java code. The challenges are normally caused by different semantic characteristics between domain-specific modeling languages and their envisaged implementation platforms. Some constructs of domain-specific modeling languages may be difficult and even laborious to implement in the target programming languages. For instance, synchronous channels of SLCO have no direct counterparts in Java. Moreover, domain-specific modeling languages are often not designed to address all details needed

when generating executable code from them, leaving some decisions to be made at the generation phase. We provide the implementation of a model-to-code transformation from SLCO models to Java code to address the challenges. The framework presented in this chapter for transforming SLCO models has first of all been designed to cover all the aspects of SLCO. In addition to this, our aim was to implement all those aspects in such a way that high performance parallel programs can be obtained. However, addressing both concerns is challenging. In Section 4.6, we further elaborate on this.

In MDSE, models and model transformations play a vital role in current software engineering practice [88]. As a consequence, powerful and efficient model management tools are needed during the development of DSMLs for the specification of models and transformations. As mentioned in Chapter 2, Epsilon [71] appears to be well-suited for many common model management tool chains, which facilitates the development of the entire transformation chain of SLCO models. Moreover, the component of the Epsilon platform for model-to-text transformation, i.e., the Epsilon Generation Language (EGL), has support for structuring model transformations by packaging transformation rules into modules. In this way, the model transformation from SLCO to Java can be systematically structured into small isolated modules and every module can be developed separately. Hence, the modification of one module will not affect the other modules. Therefore, the transformation from SLCO models to Java code in this chapter is defined via EGL.

Besides model-to-code transformations, code transformed from models can also be constructed in a modular way. In contrast to one monolithic chunk of tightly-coupled code in which every unit may interface with any other, the generated code can be composed of smaller, separated chunks that are well isolated. In this way, the maintainability and re-usability of the generated code are increased. The target language Java, as an object-oriented programming language, supports encapsulation and information hiding by providing interfaces, abstract classes, and inheritance. These features make it possible to divide the generated code into small isolated chunks. We make a distinction between *model-generic* concepts and *model-specific* parts of SLCO models. Correspondingly, Java code is also constructed by combining two parts: generic code and specific code. Generic code corresponds to model-generic concepts of SLCO models while specific code is constructed from model-specific parts of SLCO models. In this way, the implementations of generic concepts can be updated without affecting the overall transformation. Besides this, generic concepts of new constructs can be added as separated chunks of code without affecting other generic concepts. In our approach we achieve modularity by firstly distinguishing between generic and specific code and secondly modularizing the generic code.

The remainder of this chapter is organized as follows: Section 4.2 describes the overall architecture of the automated transformation from SLCO to Java code. Sections 4.3 to 4.5 demonstrate implementations of primitives of SLCO models in Java, such as state machines, statements as well as channels. Section 4.7 summarizes this chapter.

4.2 Framework Architecture

The transformation chain of SLCO models consists of multiple steps. The research on the preceding steps including several model-to-model transformations from SLCO models to more refined SLCO models is presented in [41]. The transformations are used to deal with potential semantic and platform differences. After these problems have been resolved, the

last step from SLCO models to executable code needs to be applied. In this thesis, we focus on the last step that transforms SLCO models to Java code, which is implemented in EGL [71] using Eclipse.

As mentioned in Section 2.1, the metamodel for SLCO is implemented using the Eclipse Modeling Framework (EMF) [1]. EMF provides a four-layer metamodelling architecture, i.e., the metameta layer (M_3), the meta layer (M_2), the model layer (M_1), and the object layer (M_0). Models on layer M_1 are used to describe real-world objects on layer M_0. A model always conforms to a unique metamodel on layer M_2 which describes the elements of the model as well as their attributes and interrelations. Analogously, a metamodel always conforms to a metametamodel on the layer M_3 which defines the elements of the metamodel as well as their attributes and the way they interrelate. A metametamodel is always expressed using the concepts and relations that it defines itself. The four-layer metamodeling architecture of EMF is depicted in Figure 4.1.

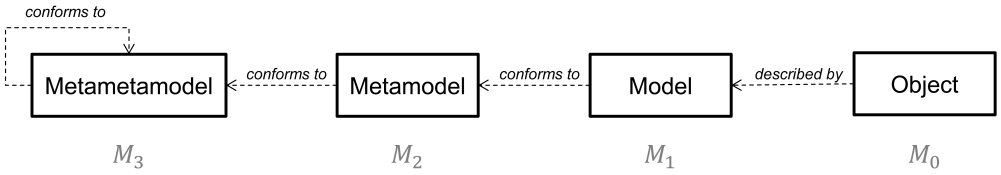

Figure 4.1: Four-layer metamodeling architecture

Model transformations used for transforming models into other formalisms can also be considered as models [103]. Correspondingly, the architecture of a model transformation can be depicted along the aforementioned four-layer metamodeling architecture as well. Therefore, following the four-layer architecture in Figure 4.1, we depict the overall architecture of our automated transformation, including the source model (SLCO), target platform (Java), and the transformation (EGL), as shown in Figure 4.2. In this figure, we do not depict the top layer M_3 which provides metametamodels for defining SLCO, Java and EGL, as the metametamodels are not relevant to define transformations on layer M_1. On layer M_2, SLCO, Java and EGL are defined and developed as metamodels. Actually, the Java language specification (JLS) itself does not provide a formal metamodel of Java. On this layer we consider the syntax of the Java language instead of a formal metamodel of Java. Based on the metamodels on layer M_2, SLCO model instances, Java programs, and the transformation definition are defined on layer M_1. The M_0 layer in this figure represents dynamic behaviors of SLCO models, run-time behavior of Java programs, and run-time instances of the transformation, respectively.

The transformation definition on the layer $M1$ in Figure 4.2 is composed of a number of templates. Each template consists of several transformation rules which describe how one or more elements in SLCO can be transformed to one or more elements in Java. All transformation rules refer to metamodels of SLCO and Java on the layer M_2 instead of SLCO instances and Java programs on the layer M_1. In this way, transformation rules are applicable for all instances of SLCO. All EGL templates for defining the transformation from SLCO models to Java code are shown in Figure 4.3. The template `Transformation.egl` file in this figure is the default when starting to execute the transformation. This file has several imported EGL templates: the `SLCO2Java.egl` is used to generate a Java class file containing the `main()` method for starting the Java program corresponding to

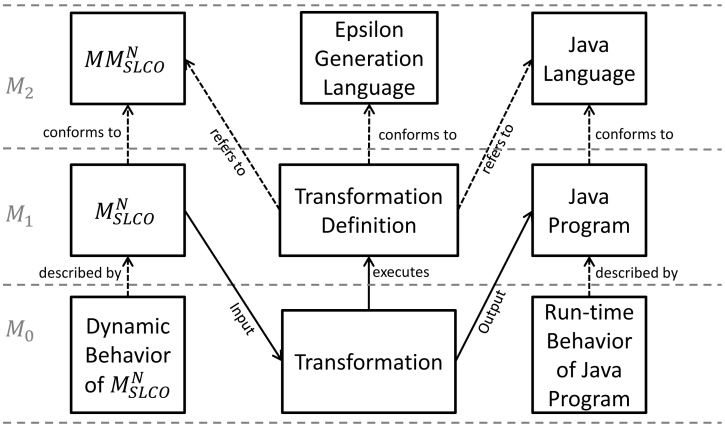

Figure 4.2: Architecture of the transformation from SLCO to Java

an SLCO model; other templates are used to define transformation rules for transforming model-dependent parts of SLCO constructs to corresponding Java class files. For example, each `Assignment` statement in an SLCO model is transformed to a separated Java class file via the template `Assignment.egl`.

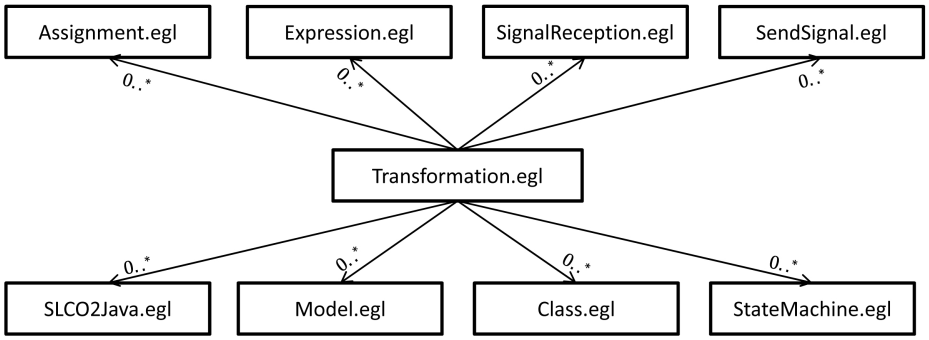

Figure 4.3: EGL templates of the transformation

To improve understandability, modifiability as well as re-usability of the implementation of SLCO models, we make a distinction between model-generic concepts and model-specific parts of SLCO models and construct a Java program by combining these two parts: generic code and specific code. Generic concepts of SLCO models are transformed into generic Java code, while aspects that are specific for concrete SLCO models are transformed into specific Java code. As a consequence, modification of some constructs of the generic code will not affect the transformation and other constructs of the generic code. Moreover, each construct of the generic code can be understood in isolation which benefits the overall understanding of the transformation. Furthermore, the existing constructs in the generic code can be reused as much as possible when the need for new target platforms arises, e.g., C++ code is required instead of Java code.

The activity diagram to derive executable code from an SLCO model is shown in Figure 4.4. The transformation from SLCO models to Java code applies transformation

rules to all the meta-model objects, which results in the generation of specific Java code corresponding to model-specific parts of an SLCO model. This specific Java code is combined with the generic Java code to obtain complete, executable code that should behave as the SLCO model specifies.

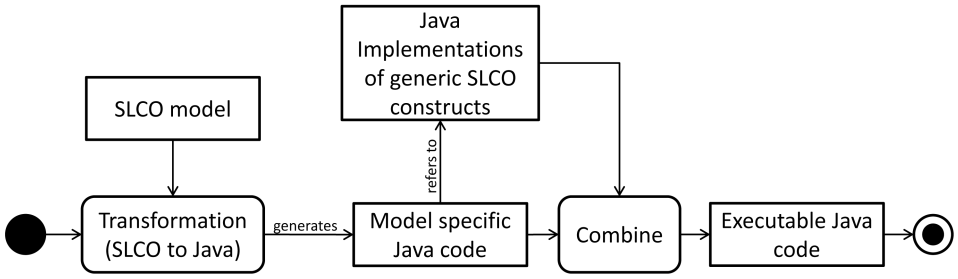

Figure 4.4: Activity diagram of the transformation process from SLCO to Java

The class diagram of the Java implementations of generic SLCO constructs is depicted in Figure 4.5. In this figure, all classes are wrapped in the package `GenericCode`. Class `Model` corresponds to the metaclass `Model` in the SLCO metamodel. It consists of a number of objects and channels which are instances of class `Class` and class `Channel`, respectively. Class `Class` is the implementation of the SLCO `Class` construct which describes the structure and behavior of its instances. Class `Channel` is an abstract class, including two abstract methods, i.e., `send` and `receive` that are implemented in its subclasses `SynchronousChannel` and `AsynchronousChannel`. Class `SynchronousChannel` implements the SLCO synchronous channels while class `AsynchronousChannel` implements the SLCO asynchronous channels. The structure of signals passed over channels is described by class `SignalMessage`, including the name of a signal as well as a list of arguments of the signal. Class `Port` in Java corresponds to the SLCO `Port` construct which connects channels to objects, i.e., instances of class `Class`. Via ports of an object, state machines in this object can send and receive signals over channels to state machines in other objects. On the Java side, each concrete SLCO state machine is defined as a subclass of abstract class `StateMachine` in the package `GenericCode`. The subclasses of abstract class `Statement`, i.e., `Assignment`, `ReceiveSignal`, `SendSignal`, and `BooleanExpression`, as shown in the figure, represent different SLCO statement types. Each SLCO statement may involve several class variables which are shared by different state machines in the same object. Therefore, we provide class `SharedVariable` to wrap the structure of SLCO class variables and class `SharedVariableList` to store all class variables involved in each SLCO statement.

In the rest of this chapter, we present the implementation of essential classes in the package `GenericCode`. In addition, we also discuss the specific Java code generated from the SLCO model example introduced in Section 2.1, referring to the classes in the package `GenericCode`.

4.3 State Machines

The most common strategies for implementing state machines are `State Pattern` [105], `Table-driven Approach` [119] and `Switch/Case Loop`. In the `State Pattern`

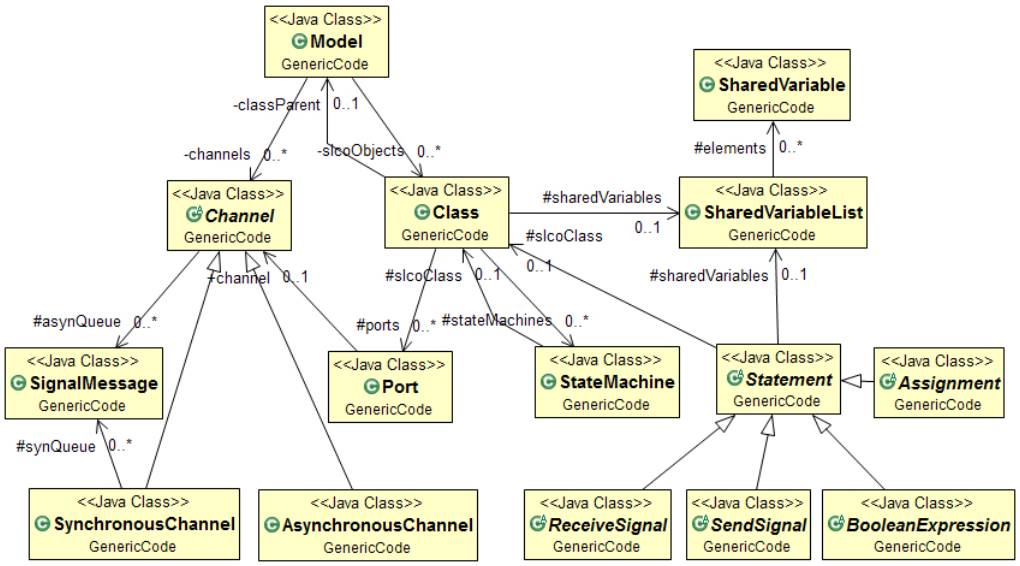

Figure 4.5: Class diagram of generic code

approach, states are represented as subclasses implementing a common interface, and each method in the interface corresponds to an event. This approach is flexible and elegant. However, it is difficult to review the structure of a state machine as its implementation is across multiple classes, especially in cases where state machines have a large number of states. In the `Table-driven Approach`, states are on one axis and events on the other axis of a table. In the cross field of the axis, one defines the actions (if there are any). This approach has the advantage that the whole state machine can be viewed in a single place making it easier to review and maintain. However, when more complicated actions are used, more complex structures are required for the representation of state table entries. The `Switch/Case Loop` is a simple and popular method to apply in applications due to its fast execution. Moreover, the structure of a state machine in the implementation is easily reviewed by developers, which facilitates the understandability of the implementation.

Therefore, we use the simple `Switch/Case Loop` approach to represent the behavior of a state machine. Each case corresponds to one state of the state machine and comprises a sequence of statements corresponding to the actions of a related outgoing state transition. If a state has more than one transition, nested switch statements are added in the generated code with a separate case for each transition.

SLCO state machines are model dependent concepts, since their structure differs from one model to another. Therefore, the corresponding Java code for the behavior of each state machine is automatically generated from SLCO models. In the package `GenericCode` we provide an abstract class `StateMachine` which only contains a constructor to initialize its field `slcoClass`, as shown in Listing 4.1. `slcoClass` is an instance of class `Class` that consists of this state machine. Each concrete state machine in an SLCO model is implemented as a subclass of class `StateMachine`.

Listing 4.1: Class StateMachine

```
1 package GenericCode;
2 public abstract class StateMachine {
```

```
3   protected Class slcoClass;
4   public StateMachine(Class slcoClass){
5     this.slcoClass = slcoClass;
6   }
7 }
```

As state machines represent concurrent processes, each state machine is mapped to a different thread in the Java implementation. This is achieved by implementing each state machine as an implementation of Java interface `Runnable`. For instance, part of the implementation of state machine `Com` depicted in Figure 2.3 is shown in Listing 4.2. The local variable `s` of state machine `Com` is transformed to an attribute of its corresponding Java class `Com`, as shown at line 2. At line 4, the constructor of class `Com` invokes the constructor of its superclass `StateMachine`, which initializes the variable `slcoClass`. At line 8, the variable `currentState` is introduced to indicate the current state of state machine `Com` during its execution. As the initial state of state machine `Com` is `Com0`, `currentState` is initialized to "Com0" at line 8. A switch loop is introduced (lines 9-34) and each case corresponds to one state of state machine `Com`. In case "Com0", a nested switch statement is added (lines 17-24). This is because state `Com0` has two outgoing transitions, i.e., one from state `Com0` to state `Com1` and the other from state `Com0` to state `Com2`. Each case of this nested switch loop corresponds to one outgoing transition of state `Com0`. For instance, "Com02Com1" in line 18 represents the transition from state `Com0` to state `Com1`. At line 13, a `String` variable named `nextTransition` is defined and is used as the nested switch statement's expression. At line 14, a boolean variable `isExecuted` is introduced to indicate whether one of these two outgoing transitions is enabled. Its initial value is **false**. When one of the two outgoing transitions is enabled, it becomes **true**.

Listing 4.2: Part of generated code for state machine Com

```
1 public class Com extends StateMachine implements Runnable {
2   protected String s;
3   public Com(Class slcoClass) {
4     super(slcoClass);
5     s = "";
6   }
7   public void run() {
8     String currentState = "Com0";
9     while(true){
10      switch(currentState){
11        case "Com0":
12          ...
13          String nextTransition;
14          boolean isExecuted = false;
15          while(!isExecuted) {
16            ...
17            switch(nextTransition) {
18              case "Com02Com1":
19                ...
20                break;
```

```
21                  case "Com02Com2":
22                    ...
23                    break;
24                  }
25                }
26                break;
27              case "Com1":
28                ...
29                break;
30              ...
31              case "Com2":
32                return;
33          }
34      }
35    }
36  }
```

As discussed in Chapter 3, non-deterministic choice between transitions is a challenging issue when implementing SLCO models at lower level languages, as the programming languages are essentially deterministic. Although mimicking non-determinism in Java is possible, not all of the solutions work efficiently. Among all solutions discussed in Section 3.3, the solution using a pseudo-random number generator to select one of outgoing transitions randomly without distinguishing between enabled and blocked transitions seems to work most efficiently. Therefore, this solution is applied in our framework to mimic non-determinism.

We explain how a pseudo-random number generator is used to mimic non-deterministic choice between SLCO transitions. In Java, class `Random` is used to generate a stream of pseudo-random numbers and its method `nextInt(int` *bound*`)` returns a pseudo-random uniformly distributed int value between 0 (inclusive) and *bound* (exclusive). This method is used to mimic non-deterministic choice among multiple outgoing transitions in our framework. More specifically, we create an array `transitions` to store names of all outgoing transitions from a state. Its length is passed as the argument of method `nextInt` when invoking it. As a result, it returns a random int value between 0 (inclusive) and the length of `transitions` (exclusive). Since the return int value can be considered as an index of the array `transitions`, the corresponding element can be obtained and assigned to the expression of the switch statement. The value of the expression is compared with each of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed.

The solution above can be applied when implementing SLCO states which have multiple outgoing transitions. For instance, in Figure 2.3 state `Com0` has two outgoing transitions. One transition from state `Com0` to state `Com1`, is associated with a statement **send** $P(true)$ **to** *Out1* and the other one, from state `Com0` to state `Com2`, is associated with a Delay statement **after** 5 **ms**. The corresponding generated Java code for state `Com0` is shown in Listing 4.3. Each case of the nested switch statement (lines 11-28) corresponds to one outgoing transition. The sequence of statements (lines 13-20) following the case `"Com02Com1"` (line 12) matches with the SLCO statement **send** $P(true)$ **to** *Out1* and the sequence of statements (lines 22-27) following the case `"Com02Com2"` (line 21) is the

Java implementation of the SLCO Delay statement **after** 5 **ms**. As previously mentioned, the variable `isExecuted` at line 3 is introduced to indicate whether one of these two outgoing transitions is enabled. Its value becomes `true` when statements associated with the outgoing transitions are enabled, as shown at lines 17 and 23, respectively. As a result, the execution of state machine `Com` exits the while loop.

At line 4, a string variable `nextTransition` is introduced as the expression of the nested switch statement. At line 5, an array `transitions` is defined and initialized with strings which indicate outgoing transitions from state `Com0`. For instance, the first element `Com02Com1` represents the transition from state `Com0` to state `Com1`. In each iteration of the while loop, a random int value is generated via a call of method `nextInt` of `rnd` (an instance of class `Random`) at line 9. This int value is assigned to the variable `idx`. The value of the element whose index is `idx` is assigned to the expression of the nested switch statement at line 10. The updated value of `nextTransition` is compared with each of the literal values in the case statements (lines 12 and 21). Consequently, the sequence of statements following the matching case is executed.

As the transition from `Com0` to `Com2` is associated with an SLCO Delay statement, we need to record the current time immediately after state machine `Com` enters case `"Com0"`. Based on this recording, we are able to check whether 5ms have passed immediately after state machine `Com` entered case `"Com02Com2"`. To this end, at line 2 we use Java method `System.currentTimeMillis()` to get the current time in milliseconds and store it in the variable `start`. At line 22, we recheck the current time via method `System.currentTimeMillis()`. If `System.currentTimeMillis()-start` is greater than 5 ms (i.e, the transition from state `Com0` to state `Com2` is enabled), then `isExecuted` becomes **true** at line 23 and the current state of state machine `Com` is set to `Com2` at line 24. Correspondingly, as the condition of the while loop (`!isExecuted`) becomes **false**, the program control passes to the line immeditately following the loop, i.e., the **break** statement at line 30. As the current state of state machine `Com` is `Com2`, then the sequence of statements following the case `"Com2"` of the outer switch statement in Listing 4.2 is executed.

Listing 4.3: Part of generated code for non-deterministic choice between transitions

```
1  case "Com0":
2    long start = System.currentTimeMillis();
3    boolean isExecuted = false;
4    String nextTransition;
5    String[] transitions = {"Com02Com1","Com02Com2"};
6    int idx;
7    Random rnd = new Random();
8    while(!isExecuted) {
9      idx = rnd.nextInt(transitions.length);
10     nextTransition = transitions[idx];
11     switch(nextTransition) {
12       case "Com02Com1":
13         SharedVariableList s_Com02Com1 = new
               SharedVariableList();
14         boolean isExe_Com02Com1 = false;
15         isExe_Com02Com1 = slcoClass.getPortName("port_Out1")
```

```
            .channel.send(new SendSignal_Com02Com1(slcoClass,
             s_Com02Com1, this), "Com02Com1", true);
16        if(isExe_Com02Com1) {
17          isExecuted = true;
18          currentState = "Com1";
19        }
20        break;
21      case "Com02Com2":
22        if((System.currentTimeMillis() - start) >= 5) {
23          isExecuted = true;
24          currentState = "Com2";
25          System.out.println("Transition: Com02Com2");
26        }
27        break;
28    }
29  }
30  break;
```

4.4 Statement

As discussed in Section 2.1, SLCO offers five types of statements: SendSignal, ReceiveSignal, (Boolean) Expression, Assignment, and Delay. The corresponding Java code for each Delay statement is automatically generated from SLCO models. For instance, a quite straightforward sequence of Java statements (lines 22-26) in Listing 4.3 is transformed from the Delay statement **after** 5 **ms**. Except for Delay, all other types of statements may involve shared variables which are accessed and modified by multiple state machines in one object, complicating the transformation. The synchronization construct to protect shared variables can be extracted from concrete statements as a generic part of the implementation. In this section, we first demonstrate this synchronization construct and then discuss the implementations of (Boolean) Expression and Assignment statements. As SendSignal and ReceiveSignal statements are relevant to channels, their corresponding implementations are discussed when presenting the implementation of channels in Section 4.5.

4.4.1 Synchronization Construct

In the generic Java code we define an abstract class called `Statement` which provides the implementation of the synchronization construct, as shown in Listing 4.4. As all subclasses of class Statement can inherit its fields and methods, we implement all SLCO statements which may involve shared variables as subclasses of class `Statement`.

As we discussed in Section 3.2, the way in which shared variables are protected has a significant impact on the overall performance of concurrent programs. In the current implementation, we choose a fine-grained locking mechanism to protect shared variables, i.e., each element in the list of shared variables involved in a statement has its own lock for read and write access. Using this fine-grained locking mechanism, an SLCO statement is transformed to a block of Java instructions with equivalent observable behavior but where the strong atomicity notion is replaced with a weaker one in Java, maintaining efficiency.

However, the aforementioned fine-grained locking mechanism may introduce deadlocks, if not carefully designed. Deadlocks arise when circular waiting occurs in a set of threads, i.e., each thread waits for another one to release a lock. This can be prevented using the technique of ordered locking, i.e., the locks are always acquired in a certain fixed order. This guarantees that when multiple threads compete over a set of variables, one thread is always able to acquire access to all of them. Therefore, a locking order is applied to the fine-grained locking mechanism.

The verification of atomicity preservation using the fine-grained locking mechanism and its deadlock freedom is presented in Chapter 5. Here, we only demonstrate its implementation. As shown in Listing 4.4, the implementation of fine-grained locking includes two methods, i.e, `lockV` at lines 4-8 and `unlockV` at lines 9-13. The methods `lockV` and `unlockV` are used to acquire and release the lock of each shared variable in the shared variables list `variablesList` which is an instance of class `SharedVariableList`. The class `SharedVariableList` is introduced to store shared variables. The elements of list are shared variables involved in a single statement and they are automatically sorted in an alphabetical order during the transformation. Correspondingly, the locks of the elements in `variablesList` are acquired in an alphabetical order when the method `lockV` is invoked.

Listing 4.4: Class Statement

```
1 public abstract class Statement {
2   protected SharedVariableList variablesList;
3   ...
4   public void lockV() {
5     for (int i = 0; i < variablesList.size(); i++) {
6       variablesList.get(i).lock.lock();
7     }
8   }
9   public void unlockV() {
10    for (int i = 0; i <= variablesList.size(); i++) {
11      variablesList.get(i).lock.unlock();
12    }
13  }
14 }
```

We use a wrapper class for `SharedVariableList` to hide information of concrete Java data structures, which facilitates modular verification when verifying the correctness of its caller. The reason for this regarding verification is explained in detail in Chapter 5. Listing 4.5 shows its implementation, including three operations (`size`, `get` and `add`).

Listing 4.5: Class SharedVariableList

```
1 public final class SharedVariableList {
2   protected ArrayList <SharedVariable> elements;
3   public SharedVariableList(){
4     elements = new ArrayList<SharedVariable>();
5   }
6   public int size(){
7     return elements.size();
```

```
8     }
9     public SharedVariable get(int index){
10      return elements.get(index);
11    }
12    public boolean add(SharedVariable v){
13      return elements.add(v);
14    }
15  }
```

Each shared variable is an instance of class `SharedVariable` which has three fields `name`, `value` and `lock`, as shown in Listing 4.6. Variables `name` (line 2) and `value` (line 3) represent the name and value of each shared variable, respectively. The lock of each shared variable can be acquired and released via the `lock()` and `unlock()` methods of Java class `Lock`, respectively. Get and set methods for variables `name` and `value` are defined, through which the variables can be accessed and new values can be assigned to them. For instance, the method `getName` (lines 5-7) is used to obtain the value that the variable `name` holds and the method `setName` (lines 8-10) is used to set a new value to the variable `name`.

Listing 4.6: Class SharedVariable

```
1   public class SharedVariable {
2   private String name;
3   private Object value;
4   protected Lock lock;
5   public String getName() {
6     return name;
7   }
8   public void setName(String name) {
9     this.name = name;
10  }
11  public Object getValue() {
12    return value;
13  }
14  public void setValue(Object value) {
15    this.value = value;
16  }
17  ...
18 }
```

4.4.2 Boolean Expressions

An SLCO boolean expression represents a statement that evaluates to either **true** or **false**. The transition associated with a boolean expression is enabled if the boolean expression evaluates to **true**. Otherwise, the execution of the transition is blocked until the boolean expression evaluates to **true**.

One generic part of implementing SLCO boolean expressions is the guarding of shared variables involved in SLCO boolean expressions through the synchronization construct

defined in Listing 4.4. Another generic part is the blocking of executions of boolean expressions when they evaluate to **false**. To implement these two generic parts, an abstract class `BooleanExpression` is introduced as a subclass of `Statement`, as shown in Listing 4.7. This class has two methods, i.e., `evaluate` (line 3) and `lockVEvaluateUnlockV` (lines 4-28). The abstract method `evaluate` is implemented in subclasses of class `BooleanExpression` regarding concrete SLCO boolean expressions. Its return value is **true** if concrete SLCO boolean expressions evaluate to **true**. Otherwise, its return value is **false**. The method `lockVEvaluateUnlockV` demonstrates how to use `lockV` and `unlockV` methods to guard shared variables involved in SLCO boolean expressions and how to mimic the blocking state of SLCO boolean expressions by using a wait-notify mechanism. The parameter of `lockVEvaluateUnlockV`, i.e., `isNonDeterTransition` (line 4), indicates whether boolean expressions are associated with non-deterministic outgoing transitions. Its value is **true** when boolean expressions are associated with non-deterministic transitions. Otherwise, its value is **false**. At line 5, method `lockV` for acquiring the locks of involved shared variables is invoked before checking the value of the boolean expression via a call of `evaluate` at lines 7 and 12.

Listing 4.7: Class BooleanExpression

```
1  public abstract class BooleanExpression extends Statement {
2    ...
3    public abstract boolean evaluate();
4    public boolean lockVEvaluateUnlockV(boolean
        isNonDeterTransition) {
5      lockV();
6      if (isNonDeterTransition) {
7        boolean isExecuted = evaluate();
8        unlockV();
9        return isExecuted;
10     }
11     else {
12       while (!evaluate()){
13         unlockV();
14         //waiting for notification from other threads
15         slcoClass.lock.lock();
16         try {
17           slcoClass.c_lock.await(10, TimeUnit.MILLISECONDS);
18         } catch (InterruptedException e) {
19           e.printStackTrace();
20         } finally {
21           slcoClass.lock.unlock();
22         }
23         lockV();
24       }
25       unlockV();
26       return true;
27     }
28   }
```

```
29 }
```

If boolean expressions are associated with non-deterministic transitions whose source states have at least two outgoing transitions, the block code at lines 7-9 in Listing 4.7 is executed. At line 7, `evaluate` is called and its return value is assigned to a boolean variable `isExecuted`. At line 8, all locks are released via a call of the method `unlockV` after checking the value of a boolean expression. At line 9, the value of `isExecuted` is passed to the caller of `lockVEvaluateUnlockV`. According to this value, the caller decides whether to take the corresponding transition that is associated with the boolean expression whose value is evaluated via a call of `evaluate`. If this value is **true**, the corresponding transition is taken. Otherwise, the caller selects another outgoing transition from the same state and then checks whether its associated statement is enabled. The solution of mimicking the non-deterministic choice between transitions is demonstrated in Section 4.3.

If boolean expressions are associated with deterministic transitions, i.e., the source state of each transition only has one outgoing transition, the `while` loop (lines 12-24) in Listing 4.7 is executed. In this case, callers of `lockVEvaluateUnlockV` need to be blocked until values of boolean expressions are **true** when executing the code within the method `lockVEvaluateUnlockV`. As discussed in Section 3.2, each blocking transition with its associated statement in SLCO can be implemented as a blocking Java block. The blocking blocks in Java can be achieved by using a busy-waiting loop or a wait-notify mechanism. The former is simple while the latter is more efficient. We therefore provide a solution based on the wait-notify mechanism using the Java interface `Condition` which provides a means for one thread to suspend its execution (to "wait") until notified by another thread. As a boolean expression may involve multiple shared variables of an SLCO object, any change of each involved shared variable may update the value of this boolean expression. Thus, we introduce a `Condition` object for all shared variables of an SLCO object, namely `c_lock`. Once the value of a boolean expression is **false**, the `await` method of the corresponding `c_lock` is called and then the current thread suspends its execution until this condition becomes **true**. If any of the shared variables that belong to the same SLCO object are updated, the `notifyAll` method of the `c_lock` is called. As a result, threads that are waiting on this condition are all woken up and then are able to re-check the corresponding boolean expressions. Since a `Condition` object is intrinsically bound to a `Lock` object, we introduce a `Lock` object for all shared variables of an SLCO object as well, called `lock`. Before waiting on a condition, the corresponding lock must be held by the current thread. As shown at line 15 in Listing 4.7, the lock is acquired by using its `lock()` method before the method `await` of its associated condition `c_lock` is invoked to suspend the execution of the current thread at line 17. As shared variables involved in boolean expressions are only updated via a call of the method `assign` (at line 6 in Listing 4.10 of class `Assignment`), the corresponding notifying part is added immediately after the method `assign` call, as shown at line 11 in Listing 4.10.

Two issues of the wait-notify mechanism discussed in Section 3.3, i.e., spurious wakeups and missing signals, are also addressed. The first issue is fixed by nesting each call of `await` inside a while loop spanning lines 12-24 in Listing 4.7. If a waiting thread wakes up without receiving a signal, the while loop will execute once more, causing this thread to go back to wait. The second issue is addressed by adding a timeout time to an `await` call, as shown in line 17. The method `await` with timeout information causes the current

thread to wait until either another thread invokes the `notify` method or the `notifyAll` method on the same object, or the specified amount of time has elapsed. In this way, a waiting thread can always wake up and re-check the condition of the while loop when the signal to wake up is missed.

As mentioned above, we introduce a `Condition` object and a `Lock` object for each SLCO object, namely `c_lock` and `lock`. Each SLCO object is mapped to an instance of class `Class` which corresponds to the SLCO `Class` construct. As shown in Listing 4.8, variables `c_lock` and `lock` are defined as fields of class `Class` at line 3 and 4, respectively. Each SLCO object consists of variables, ports as well as state machines. At line 5, a shared variable list `sharedVariables` is defined for storing all variables of an SLCO object. The ports and state machines of an SLCO object are stored in lists `ports` (line 6) and `stateMachines` (line 7), respectively.

As discussed in Section 3.7, one challenge when transforming SLCO models to good implementations is to ensure the fairness between threads in the implementation to some extent by using the right constructs obtained from Java library code. We therefore consider fairness using the optional fairness policy that is provided by locks in Java. Locks in Java support an optional fairness policy. For instance, the constructor for class `ReentrantLock` accepts an optional fairness parameter. When it is set to **true**, locks favor granting access to the longest-waiting thread. Note that fairness for locks does not guarantee fairness of thread scheduling which is an independent other issue. Therefore, fairness for locks does prevent starvation of threads only to a limited extent. At line 13, `lock` is initialized with an instance of class `ReentrantLock` and the parameter of its constructor is set to **true**. At line 14, the condition `c_lock` is bound to the lock `lock` by using its `newCondition()` method.

Listing 4.8: Part of generic code for SLCO Class

```
1  public class Class {
2    ...
3    protected Condition c_lock;
4    protected Lock lock;
5    protected SharedVariableList sharedVariables;
6    protected ArrayList<Port> ports;
7    protected ArrayList<StateMachine> stateMachines;
8    public Class(String className, String objectName, Model
       parent) {
9      ...
10     sharedVariables = new SharedVariableList();
11     ports = new ArrayList<Port>();
12     stateMachines = new ArrayList<StateMachine>();
13     lock = new ReentrantLock(true);
14     c_lock = lock.newCondition();
15   }
16   ...
17 }
```

The specific part of an SLCO boolean expression is implemented as a subclass of class `BooleanExpression` in Listing 4.7. For instance, the transition from `SendRec0` to `SendRec1` in Figure 2.3 is associated with a boolean expression $m == 6$ and m is

an integer variable shared by three state machines `Rec1`, `Rec2` and `SendRec`. The specific part of the boolean expression $m == 6$ is shown in Listing 4.9. The method `evaluate` at lines 4-6 is the implementation of the abstract method `evaluate` of class `BooleanExpression` at line 3 in Listing 4.7. The boolean value **true** is returned to its caller when the value of variable m equals 6. Otherwise, the boolean value **false** is returned to its caller.

Listing 4.9: Part of generated code for a specific SLCO (Boolean) Expression statement

```
1 public class BooleanExpression_SendRec02SendRec1 extends
    BooleanExpression{
2   ...
3   @Override
4   public boolean evaluate() {
5     return ((Integer)slcoClass.getSharedVariableName("m").
        getValue()==6);
6   }
7 }
```

The method `getSharedVariableName` at line 5 in Listing 4.9 returns a shared variable object whose string name is equal to `"m"`. Its value can be obtained via a call of the method `getValue` (lines 11-13 in Listing 4.6).

4.4.3 Assignment

The generic part of the class `Assignment` is defined as a subclass of class `Statement`, as shown in Listing 4.10. Its method `lockVAssignUnlockV` (lines 4-15) is used to safely assign a new value to a shared variable via method `assign` (line 3). The abstract method `assign` is implemented in subclasses which are related to concrete SLCO assignments. Whenever a thread attempts to execute the method `assign`, it needs to acquire all involved locks via a call to the method `lockV` (at line 5). After the execution of the method `assign` at line 6, all locks are released via a call to the method `unlockV` at line 7. If any threads are waiting on the condition `c_lock`, then they are woken up by a call to the `signalAll` method at line 11. The notifying part at lines 9-14 matches the waiting part at lines 15-22 in Listing 4.7. Before signaling waiting threads on the condition, the current thread needs to acquire the lock that is associated with the condition `c_lock`, as shown at line 9. After signaling waiting threads on the condition, the current thread needs to release the lock as well, as shown at line 13.

Listing 4.10: Class Assignment

```
1 public abstract class Assignment extends Statement {
2   ...
3   public abstract void assign();
4   public void lockVAssignUnlockV() {
5     lockV();
6     assign();
7     unlockV();
8     //notifying all waiting threads on Lock lock
9     slcoClass.lock.lock();
```

```
10      try {
11        slcoClass.c_lock.signalAll();
12      } finally {
13        slcoClass.lock.unlock();
14      }
15    }
16  }
```

The specific part of an SLCO assignment is implemented as a subclass of class `Assignment`. For instance, the implementation of the concrete assignment $m := m + 1$ associated with the transition from state `Rec2b` to state `Rec2a` of state machine `Rec2` in Figure 2.3 is shown in Listing 4.11. The method assign (lines 4-6) is the implementation of the abstract method inherited from its parent `Assignment` in Listing 4.10.

Listing 4.11: Part of generated code for an SLCO Assignment statement

```
1 public class Assignment_Rec2b2Rec2a extends Assignment{
2   ...
3   @Override
4   public void assign() {
5     slcoClass.getSharedVariableName("m").setValue(((Integer)
          slcoClass.getSharedVariableName("m").getValue()+1));
6   }
7 }
```

4.5 Channels

As mentioned in Section 2.1, SLCO supports synchronous channels, asynchronous lossless channels as well as asynchronous lossy channels for communication between different objects. They are either bidirectional or unidirectional. As lossy SLCO channels are an undesired aspect of physical connections and a bidirectional channel can be easily replaced with two unidirectional channels via model-to-model transformations, in this section we only focus on the implementation of unidirectional lossless SLCO channels which are associated with only one-place buffers.

We define an abstract class `Channel` which contains three abstract methods: `send` (at line 12), `put` (at line 13), and `receive` (at line 14), as shown in Listing 4.12. The asynchronous and synchronous SLCO channels are implemented as its subclasses which create full implementations of these methods with specialized behaviors regarding asynchronous and synchronous communication in SLCO. The one-place buffer, as discussed in Section 3.4, can be implemented using the implementation of Java interface `BlockingQueue`, i.e., class `ArrayBlockingQueue`. Therefore, at line 2 an instance of class `ArrayBlockingQueue` called `asynQueue` is declared and is initialized with capacity one in its subclasses `AsynchronousChannel` (Listing 4.18) and `SynchronousChannel` (Listing 4.24), respectively. Elements to be stored in `asynQueue` are instances of class `SignalMessage` which is defined in Listing 4.13. The class `SignalMessage` provides the structure of signals passed over channels.

Listing 4.12: Class Channel

```
1 public abstract class Channel {
2   protected ArrayBlockingQueue<SignalMessage> asynQueue;
3   protected Lock receiverLock;
4   protected Lock senderLock;
5   protected Condition c_receiverLock;
6   public Channel() {
7     // asynQueue = new ArrayBlockingQueue<SignalMessage>(1);
8     receiverLock = new ReentrantLock(true);
9     senderLock = new ReentrantLock(true);
10    c_receiverLock = receiverLock.newCondition();
11  }
12  public abstract boolean send(SendSignal sendSignal, String
        transitionName, boolean isNonDeterTransition);
13  public abstract boolean put(SignalMessage signal);
14  public abstract SignalMessage receive(ReceiveSignal
       receiveSignal, String transitionName, boolean
       isNonDeterTransition);
15  public SignalMessage peek() {
16    SignalMessage signal = asynQueue.peek();
17    return signal;
18  }
19 }
```

The send and receive operations of an SLCO channel are implemented by using several operations of class `ArrayBlockingQueue`. Although each operation of class `ArrayBlockingQueue` is thread-safe, we still need to introduce locks when implementing operations of channels. This is because the SLCO send or receive operations of a channel are mapped to a block of Java statements which involves more than one operation of class `ArrayBlockingQueue`. For instance, the block of statements for implementing the receive operation of a channel consists of retrieving the signal in the corresponding buffer but not removing it, checking the content of the signal, and then removing the signal from the buffer. The first step can be achieved by using the method `peek` of class `ArrayBlockingQueue` which retrieves the head of the queue but does not remove it from the queue and the third step can be implemented by using the method `take` of class `ArrayBlockingQueue` which retrieves and removes the head of the queue. As the receive operation of a channel in SLCO is atomic, a lock needs to be introduced to ensure the atomic execution of its corresponding block of Java statements. Therefore, at line 3 (Listing 4.12), a lock `receiverLock` is introduced to control access of the receive operation executions of multiple threads. For the same reason, a lock for the implementation of the send operation of a channel is needed as well. As shown at line 4 (Listing 4.12), a lock `senderLock` is introduced to control access of the send operation of executions of multiple threads.

At the SLCO level, if the buffer associated with a channel is full, send signal statements that send signals via this channel are blocked and the corresponding state machines are blocked as well. Such a blocking situation can be achieved in the Java implementation by using a blocking inserting operation of class `ArrayBlockingQueue`, like `put`. Similarly,

if the buffer associated with a channel is empty, signal reception statements that receive signals via this channel are blocked and corresponding state machines are blocked as well. However, this kind of situation can not be achieved in Java by using the blocking feature of retrieving operations of class `ArrayBlockingQueue`, like `take`. This is because when the method `take` is called, the execution of the corresponding state machine is suspended until a signal is available in the buffer. After a signal is available, the state machine just retrieves and removes it from the buffer without checking the content of it, which conflicts with the fact that a state machine always needs to check the name of each signal before receiving it.

As discussed in Section 3.3, the wait-notify mechanism is an efficient way to mimic the blocking state in Java. Therefore, we let state machines suspend by using the wait-notify mechanism instead of using the blocking feature of class `ArrayBlockingQueue` when signal reception statements are not enabled. Specifically, we introduce an instance of class `Condition` at line 5 in Listing 4.12, i.e., `c_receiverLock` which is bound to `receiverLock`. When a signal reception statement is not enabled, the method `await` of `c_receiverLock` is called. Then the corresponding state machine releases the lock `receiverLock` and its execution is suspended as well. On the sending side, only after the send operation of the channel is executed, the element in the buffer is updated. Therefore, the corresponding operation for waking up suspended threads needs be called after a state machine has executed the send operation of the channel. The implementation of the wait-notify mechanism is demonstrated later through implementations of asynchronous and synchronous channels.

The call of the method `peek` of `asynQueue` at line 16 in Listing 4.12 is wrapped in the method `peek` at lines 15-18 which is shared by all subclasses of class `Channel`. As mentioned before, elements in `asynQueue` are instances of class `SignalMessage` which defines the structure of signals sent over channels, as shown Listing 4.13. Each signal has a name and a list of arguments, which are defined as attributes of `SignalMessage`, i.e., `name` (line 3), and `args` (line 4), respectively. As the number of arguments of a signal is arbitrary, we define the constructor of class `SignalMessage` as a varargs Java method which takes a variable number of arguments and the variable-length argument is specified by three dots (...). As shown at line 9, when an instance of class `SignalMessage` is created, the first argument is matched with the first parameter name and the remaining arguments belong to args.

Listing 4.13: Class SignalMessage

```
1  package GenericCode;
2  public class SignalMessage {
3    public String name;
4    public Object[] args;
5    public SignalMessage(String name) {
6      this.name = name;
7      this.args = null;
8    }
9    public SignalMessage(String name, Object...args){
10     this.name = name;
11     this.args = args;
12   }
```

```
13 }
```

SendSignal Statement As `SendSignal` statements may involve shared variables accessed by different state machines, the synchronization construct defined in class `Statement` is also needed when executing this kind of statements. Therefore, we define class `SendSignal` as a subclass of class `Statement` to inherit the synchronous construct, as shown in Listing 4.14.

Listing 4.14: Class SendSignal

```
1 public abstract class SendSignal extends Statement {
2   public SendSignal(Class slcoClass, SharedVariableList
       sharedVariables) {
3     super(slcoClass, sharedVariables);
4   }
5   public abstract boolean send();
6   public boolean lockVSendUnlockV() {
7     lockV();
8     boolean isSuccessful = send();
9     unlockV();
10    return isSuccessful;
11  }
12 }
```

The send at line 5 in Listing 4.14 is defined as an abstract method, which is implemented in its subclasses regarding the model specific part of each concrete `SendSignal` statement. For instance, the implementation of the concrete `SendSignal` statement **send** $Q(5)$ **to** *Out2* associated with the transition from state `Com1` to state `Com3` in Figure 2.3 is shown in Listing 4.15. An instance of `SignalMessage` is created at line 9 according to the signal Q(5). Such a signal is then added to the corresponding buffer of the channel at line 10 by calling method `put` which is defined in Listing 4.21.

Listing 4.15: Generated code for an SLCO SendSignal statement

```
1 public class SendSignal_Com12Com3 extends SendSignal{
2   public Com sm;
3   public SendSignal_Com12Com3(Class slcoClass,
       SharedVariableList  sharedVariables, Com sm) {
4     super(slcoClass, sharedVariables);
5     this.sm = sm;
6   }
7   @Override
8   public boolean send() {
9     SignalMessage signal = new SignalMessage("Q", new Object
        []{5});
10    return slcoClass.getPortName("port_Out2").channel.put(
        signal);
11  }
12 }
```

ReceiveSignal Statement Similarly, abstract class `ReceiveSignal` for extracting the generic part of `ReceiveSignal` statements is defined as a subclass of `Statement`, as shown in Listing 4.16. The method `lockVEvaluateReceiveUnlockV()` at lines 7-21 demonstrates how to use `lockV` and `unlockV` methods to safely operate involved shared variables in `ReceiveSignal` statements. After acquiring the lock of each involved shared variable via calling `lockV` at line 8, the `evaluate` which is used to check the name of the signal and the conditional expression over the signal arguments is called. If the name is expected and the conditional expression is held, the `receive` is called to take the signal from the corresponding buffer. Moreover, a notifying part at lines 13-17 is called to wake up all threads waiting for changes of the values of shared variables, as values of shared variables may be updated via `ReceiveSignal` statements. For instance, the `ReceiveSignal` statement **receive** $Q(m|m >= 0)$ **from** *In2* changes the value of shared variable `m` if a value greater or equal to 0 is received.

Listing 4.16: Class ReceiveSignal

```
1 public abstract class ReceiveSignal extends Statement {
2   public ReceiveSignal(Class slcoClass, SharedVariableList
       sharedVariables) {
3     super(slcoClass, sharedVariables);
4   }
5   public abstract boolean evaluate();
6   public abstract void receive();
7   public boolean lockVEvaluateReceiveUnlockV() {
8     lockV();
9     boolean isExecutedStatement = evaluate();
10    if (isExecutedStatement) {
11      receive();
12      slcoClass.lock.lock();
13      try {
14        slcoClass.c_lock.signalAll();
15      } finally {
16        slcoClass.lock.unlock();
17      }
18    }
19    unlockV();
20    return isExecutedStatement;
21  }
22 }
```

The corresponding implementations of abstract methods `evaluate` (line 5) and `receive` (line 6) in Listing 4.16 for statement **receive** $Q(m|m >= 0)$ **from** *In2* is shown in Listing 4.17 at lines 4-7 and lines 9-12, respectively.

Listing 4.17: Part of generated code for an SLCO ReceiveSignal statement

```
1 public class ReceiveSignal_Rec2a2Rec2b extends ReceiveSignal
    {
2   ...
3   @Override
```

```
4   public boolean evaluate() {
5     SignalMessage signal = slcoClass.getPortName("port_In2")
        .channel.peek();
6     return ((String)signal.name).equals("Q") && signal.args.
        length == 1 && ((Integer)(signal.args[0])) >= 0;
7   }
8   @Override
9   public void receive() {
10    SignalMessage signal = slcoClass.getPortName("port_In2")
        .channel.peek();
11    slcoClass.getSharedVariableName("m").setValue((Integer)
        signal.args[0]);
12  }
13 }
```

4.5.1 Asynchronous Channels

The model-generic part of SLCO's asynchronous channels is implemented as a subclass of the abstract class Channel (Listing 4.12), as shown in Listing 4.18. The associated buffer for each asynchronous channel is implemented using ArrayBlockingQueue with a buffer capacity of 1, i.e., asynQueue initialized at line 3. The implementations of send and receive operations of an asynchronous channel are based on put and take operations of ArrayBlockingQueue, as shown in Listings 4.19 and 4.22, respectively.

Listing 4.18: class AsynchronousChannel

```
1 public class AsynchronousChannel extends Channel {
2   public AsynchronousChannel() {
3     asynQueue = new ArrayBlockingQueue<SignalMessage>(1);
4   }
5   @Override
6   public boolean send(SendSignal sendSignal,
7   String transitionName, boolean isNonDeterTransition) { ...
        }
8   @Override
9   public boolean put(SignalMessage signal) { ...}
10  @Override
11  public SignalMessage receive(ReceiveSignal receiveSignal,
12  String transitionName, boolean isNonDeterTransition) { ...
        }
13 }
```

The Implementation of the send Method The implementation of the send operation of the asynchronous channel is shown in Listing 4.19.

When executing an SLCO SendSignal statement and the corresponding buffer is empty, a signal will be added to the buffer and then the method send returns `true` to its caller, i.e., the corresponding state machine, as shown at lines 6-15. Otherwise, the method send

returns `false` to the caller, as shown at lines 16-18. Note that when the send operation is executed successfully, a notifying part for waking up all threads suspending on receive operations of the channel is needed, as shown at lines 9-14.

If a send operation is associated with a deterministic transition, it blocks as well as the execution of the corresponding state machine blocks when the buffer is full. This is achieved by the semantics of the blocking put operation of class ArrayBlockingQueue. After adding the signal to the buffer successfully, the notifying part for waking up all threads suspending on receiving operations of the channel is executed at lines 22-27.

The method lockVSendUnlockV called at lines 7 and 20 in Listing 4.19 is defined at lines 6-11 in abstract class `SendSignal` in Listing 4.14.

Listing 4.19: The send method of class AsynchronousChannel

```
1 public boolean send(SendSignal sendSignal, String
    transitionName, boolean isNonDeterTransition) {
2   try{
3     senderLock.lock();
4     SignalMessage signal = peek();
5     if (isNonDeterTransition) {
6       if (signal == null) {
7         sendSignal.lockVSendUnlockV();
8         System.out.println("Transition: " + transitionName);
9         try {
10          receiverLock.lock();
11          c_receiverLock.signalAll();
12        } finally {
13          receiverLock.unlock();
14        }
15        return true;
16      } else {
17        return false;
18      }
19    } else {
20      sendSignal.lockVSendUnlockV();
21      System.out.println("Transition: " + transitionName);
22      try {
23        receiverLock.lock();
24        c_receiverLock.signalAll();
25      } finally{
26        receiverLock.unlock();
27      }
28      return true;
29    }
30  } finally {
31    senderLock.unlock();
32  }
33 }
```

In the generated implementation of a state machine, the method `send` (Listing 4.19) is called when sending a message over a channel. For instance, the transition from state `Com1` to state `Com3` in Figure 2.3 is associated with a `SendSignal` statement **send** $Q(5)$ **to** *Out2* and the specific part of its implementation is automatically generated from the transformation, as shown in Listing 4.20. At line 3, an instance of class `SendSignal`, i.e., `SendSignal_Com12Com3` is created and it is passed to the method `send`. The implementation of class `SendSignal_Com12Com3` is shown in Listing 4.15.

Listing 4.20: Generated code for a call of the send method

```
1 case "Com1":
2   SharedVariableList s_Com12Com3 = new SharedVariableList();
3   slcoClass.getPortName("port_Out2").channel.send(new
       SendSignal_Com12Com3(slcoClass, s_Com12Com3, this), "
       Com12Com3", false);
4   currentState = "Com3";
5   break;
```

The Implementation of the put Method The abstract method `put` at line 13 of class `Channel` in Listing 4.12 is implemented in its subclass `AsynchronousChannel`, as shown in Listing 4.21. This method wraps the put operation of class ArrayBlockingQueue which guarantees that send operations associated with deterministic transitions block when the corresponding buffer is full.

Listing 4.21: The put method of class AsynchronousChannel

```
1 public boolean put(SignalMessage signal) {
2   try {
3     asynQueue.put(signal);
4   } catch (InterruptedException e) {
5     e.printStackTrace();
6   }
7   return true;
8 }
```

The Implementation of the receive Method The implementation of the receive operation of the asynchronous channel is shown in Listing 4.22.

Due to the conditional reception feature of the ReceiveSignal statement of SLCO, the content of a signal in the buffer will only be received by a state machine if the conditional expression over the signal arguments evaluates to true. In the Java code, this requires that the element is retrieved without taking the element first. To this end, the `peek` method which is defined in Listing 4.12 method is called at line 7.

If the signal is not `null` (line 8) and the conditional expression over signal arguments evaluates to true as well (line 9), then the signal is taken via the `take` method (line 11) and the signal is returned to the caller of the method `receive` (line 13). Otherwise, the signal with the value of `null` is returned to its caller (line 17) or the receive operation will be blocked for a while (line 19) and the buffer will be checked again until the signal in the buffer satisfies the conditional expression over the signal arguments. The code

at lines 16-18 corresponds to the case where the receive operation is associated with a non-deterministic transition. The code at lines 18-20 corresponds to the case where the receive operation is associated with a deterministic transition but the current signal in the buffer does not satisfy the condition. In this case, the lock `receiverLock` is released and the current thread waits until it is signaled or the specified waiting time elapses, as shown at line 19.

Listing 4.22: The receive method of class AsynchronousChannel

```
1  public SignalMessage receive(ReceiveSignal receiveSignal,
       String transitionName, boolean isNonDeterTransition) {
2    boolean isExecuted = false;
3    SignalMessage signal = null;
4    try {
5      receiverLock.lock();
6      while (!isExecuted) {
7        signal = peek();
8        if (signal != null) {
9          isExecuted = receiveSignal.
               lockVEvaluateReceiveUnlockV();
10         if (isExecuted) {
11           signal = (SignalMessage) asynQueue.take();
12           System.out.println("Transition: " + transitionName
                 );
13           return signal;
14         }
15       }
16       if (isNonDeterTransition) {
17         return null;
18       } else {
19         c_receiverLock.await(10, TimeUnit.MILLISECONDS);
20       }
21     }
22   } catch (InterruptedException e) {
23     e.printStackTrace();
24   } finally {
25     receiverLock.unlock();
26   }
27   return null;
28 }
```

In the generated implementation of a state machine, the method `receive` (Listing 4.22) is called when receiving a message over a channel. For instance, the `receive` method is called at line 4 in Listing 4.23 which corresponds to the transition from state `Rec2a` to state `Rec2b` of the state machine `Rec2` in Figure 2.3. Since the `ReceiveSignal` statement **receive** $Q(m|m >= 0)$ **from** *In2* involves the shared variable `m`, an instance of class `SharedVariableList` is created at line 2 and this list is used to store all involved shared variables in the statement **receive** $Q(m|m >= 0)$ **from** *In2*. For instance, the shared variable `m` is added to the list at line 3.

Listing 4.23: Generated code for a call of the receive method

```
1 case "Rec2a":
2   SharedVariableList s_Rec2a2Rec2b = new SharedVariableList
       ();
3   s_Rec2a2Rec2b.add(slcoClass.getSharedVariableName("m"));
4   SignalMessage signal = slcoClass.getPortName("port_In2").
       channel.receive(new ReceiveSignal_Rec2a2Rec2b(slcoClass
       , s_Rec2a2Rec2b, this),"Rec2a2Rec2b", false);
5   currentState = "Rec2b";
6   break;
```

4.5.2 Synchronous Channels

As discussed in Section 3.4.2, the implementation of SLCO synchronous channels is challenging because of the combination of synchronous communication with non-determinism. In such a case, if a receive or send operation is not enabled, the execution of the corresponding state machine is not blocked at the receive or send operation. This kind of synchronous communication is difficult to implement, as threads that are executed in parallel do not have the global overview of the behavior of the program.

To this end, we add a handshake-like synchronization protocol between state machine threads for synchronous communication in the Java implementation. The implicit buffer associated with each SLCO synchronous channel is implemented using two kinds of Java queue from the package `java.util.concurrent`, i.e, `ArrayBlockingQueue` and `SynchronousQueue`. When a signal is sent over a synchronous channel, the signal is added to the corresponding `ArrayBlockingQueue` queue first, and then is added to the corresponding `SynchronousQueue` queue as well. In this way, a thread is able to check the content of the signal by means of reviewing the signal from the `ArrayBlockingQueue` before starting the synchronous communication via the operation of the `SynchronousQueue` queue.

As shown in Listing 4.24, the model-generic part of SLCO synchronous channel is implemented as a subclass of class `Channel`. An instance of class `SynchronousQueue`, i.e., `synQueue`, is declared at line 2 and initialized at line 5. `asynQueue` is an instance of class `ArrayBlockingQueue`, which is declared in its superclass `Channel`. At line 4, `asynQueue` is initialized with capacity one. The abstract methods of its superclass `Channel`, i.e., `send`, `put`, and `receive`, need to be implemented regarding SLCO synchronous communication between objects. The implementations are demonstrated in Listings 4.25, 4.26 and 4.27, respectively.

Listing 4.24: Class SynchronousChannel

```
1 public class SynchronousChannel extends Channel {
2   protected SynchronousQueue<SignalMessage> synQueue;
3   public SynchronousChannel() {
4     asynQueue = new ArrayBlockingQueue<SignalMessage>(1);
5     synQueue = new SynchronousQueue<SignalMessage>();
6   }
7   @Override
```

```
8   public boolean send(SendSignal sendSignal, String
       transitionName, boolean isNonDeterTransition) { ...}
9   @Override
10  public boolean put(SignalMessage signal) { ... }
11  @Override
12  public SignalMessage receive(ReceiveSignal receiveSignal,
       String transitionName, boolean isNonDeterTransition) {
       ... }
13 }
```

The Implementation of the send Method The implementation of the `send` method at line 8 in Listing 4.24 is shown in Listing 4.25. The lock `senderLock` is acquired at line 3 to prevent simultaneous access between threads of the send operation of a synchronous channel. At line 4 the `sendSignal` statement is evaluated via the method `lockVSendUnlockV`. If the `sendSignal` statement is enabled, a boolean value true is returned to the caller of the `send` method at line 7. Otherwise, a boolean value false is returned at line 9. According to this value, the caller decides whether to take the corresponding transition that is associated with the `sendSignal` statement. If this value is true, the corresponding transition is taken. Otherwise, the caller rechecks whether the `sendSignal` statement is enabled.

Listing 4.25: The send method of class SynchronousChannel

```
1 public boolean send(SendSignal sendSignal, String
     transitionName, boolean isNonDeterTransition) {
2   try{
3     senderLock.lock();
4     boolean isExecutedStatement = sendSignal.
         lockVSendUnlockV();
5     if (isExecutedStatement) {
6       System.out.println("Transition: " + transitionName);
7       return true;
8     } else {
9       return false;
10    }
11  } finally{
12    senderLock.unlock();
13  }
14 }
```

The Implementation of the put Method The implementation of the `put` method of class `SynchronousQueue` is shown in Listing 4.26.

As mentioned above, in order to let a thread be able to check the content of a signal before receiving it, the signal is first added to the `asynQueue`, as shown at line 4. Since some threads may be suspended on the receive operation of the channel, a notifying part is needed to wake all of them up, as shown at lines 5-7. After these steps, the signal is added to the `synQueue` via its `offer` method at line 8. By adding timeout information

to the call of the method `offer` at line 8, the current thread inserts a signal into the `synQueue` and waits, if necessary, up to the specified wait time, for another thread to take it. Correspondingly, the execution of the current thread will not be blocked at line 8 when the corresponding `SendSignal` statement is not enabled.

After the execution of the adding operation of `synQueue` at line 8, the signal in the `asynQueue` needs to be removed, as shown at lines 9-11. If the signal is added to the `synQueue`, i.e., the return value of the method `offer` is true, a boolean value true is returned to the caller of the `put` method. Otherwise, a boolean value false is returned.

Listing 4.26: The put method of class SynchronousChannel

```
1 public boolean put(SignalMessage signal) {
2   boolean isTaken = false;
3   try {
4     asynQueue.put(signal);
5     receiverLock.lock();
6     c_receiverLock.signalAll();
7     receiverLock.unlock();
8     isTaken = synQueue.offer(signal, 10, TimeUnit.
        MILLISECONDS);
9     receiverLock.lock();
10    asynQueue.take();
11    receiverLock.unlock();
12  } catch (InterruptedException e) {
13    e.printStackTrace();
14  }
15  return isTaken;
16 }
```

The Implementation of the receive Method The implementation of the receive operation of the synchronous channel is shown in Listing 4.27. Similar to the implementation of `receive` of asynchronous channels, the current thread executing the `receiveSignal` statement is blocked at the `while` loop at lines 4-28 when the value of `isNonDeterTransition` is false. To avoid the inefficient busy-waiting loop, the `await` method for waiting for the corresponding send operation is used at line 21. If the value of `isNonDeterTransition` is true, the while loop will be executed only once and a value will be returned to the caller of the method `receive`. It returns the `signal` to its caller if the `receiveSignal` is enabled. Otherwise, it returns `null` to its caller.

Listing 4.27: The receive method of class SynchronousChannel

```
1 public SignalMessage receive(ReceiveSignal receiveSignal,
    String transitionName, boolean isNonDeterTransition) {
2   boolean isExecuted = false;
3   SignalMessage signal = null;
4   while (!isExecuted) {
5     try {
6       receiverLock.lock();
7       signal = peek();
```

```
8         if (signal != null) {
9           isExecuted = receiveSignal.
               lockVEvaluateReceiveUnlockV();
10          if (isExecuted) {
11            signal = (SignalMessage) synQueue.poll(10,
                 TimeUnit.MILLISECONDS);
12            if (signal != null) {
13              System.out.println("Transition: " +
                   transitionName);
14              return signal;
15            }
16          }
17        }
18        if (isNonDeterTransition) {
19          return null;
20        } else {
21          c_receiverLock.await(10, TimeUnit.MILLISECONDS);
22        }
23      } catch (InterruptedException e) {
24        e.printStackTrace();
25      } finally {
26        receiverLock.unlock();
27      }
28    }
29    return null;
30  }
```

4.6 Discussion

The framework presented in this chapter for transforming SLCO models has been designed to cover all the aspects of SLCO. Our secondary concern is that high performing parallel programs should be achievable via the transformation. However, addressing both concerns is challenging, as some language aspects are hard to implement in a way favorable for high performance. As a result, the transformation of models using those aspects currently tends to not produce highly efficient programs. For instance, SLCO supports non-determinism whereas the target programming language Java is essentially deterministic. Cases combining non-determinism and potential blocking of statements, i.e., where an SLCO state has multiple outgoing transitions containing potentially blocking statements, and a non-deterministic choice has to be made between them, are challenging to transform to efficient code. Mimicking the non-deterministic choice between transitions is possible in Java by applying a pseudo-random number generator with a while loop. This solution is applied in the current framework. A drawback of this approach is that executing the while loop may lead to waiting arbitrarily long, in particular when in each loop iteration, a transition with a blocked statement is selected. This, in turn, can lead to potential delays that depend on the size of the time slices of the preemptive scheduler. In particular, the potentially long wait in a while loop can lead to low performance when the scheduler is

not preemptive.

Therefore, the model-to-code transformation presented in the current section is a proof of concept, performance-wise. However, by supporting all aspects of SLCO, we leave the option open for a developer to choose one of the following development steps. One can transform the model as it is, resulting in code that can be run to test the behavior of the application rather than only focusing on high performance. In case high performance is required, one can also transform the model via model-to-model transformations to a version that has a structure that can be straightforwardly transformed to a high performance program. For instance, as a preprocessing step, non-determinism that currently is not transformed to efficient code can be removed from the model before code is generated. Correspondingly, many performance issues can be avoided.

Finally, as an interesting and challenging step of our future research, we are currently working on a new version of the model-to-code transformation, aiming to further improve the performance of the generated programs.

4.7 Conclusions

To address several challenges we identified in Chapter 3, we defined a framework for transforming SLCO models consisting of concurrent communicating objects to multi-threaded Java programs in this chapter. By implementing different SLCO concepts in Java, we demonstrated concrete behavior gaps between domain-specific modeling languages and envisaged implementation platforms. Also we presented how to bridge these gaps by finding patterns in programming languages for correctly capturing concurrent model semantics. By defining the transformation as well as the corresponding executable Java code in a modular way, we improve the understandability and re-usability of this framework. Moreover, the automatic model-to-code transformation we defined not only reduces the cost and time associated with the manual coding effort but also eliminates possible introduction of human errors in the transformation process. For instance, the order of locks for accessing shared variables is fixed during the transformation, which helps to avoid lock-deadlocks.

Model-to-code transformations as well as code produced by them may not be free of errors, such as simple programming errors and mistakes in the transformation from model to code. To ensure the correctness of produced code and transformations themselves, a formal specification logic and proof support is required. The framework presented in this chapter, as a first attempt to fully preserving the semantics of SLCO models in the implementation, provides automatic generation of relatively realistic and efficient code, which complicates the verification. For instance, taking conditional receive operations and non-determinism into account, a complicated protocol is needed when implementing SLCO channels. Moreover, the wait-notify mechanism introduced to improve the efficiency of the corresponding concurrent code in Java complicates the verification as well. As a first step towards verifying the correctness of the code generation, we do not yet consider the wait-notify mechanism when verifying the preservation of atomicity in Chapter 5. Also, we verify a simplified version of the SLCO asynchronous channel implementation in Chapter 6. These simplifications facilitate the verification of our implementation.

Chapter 5

Verifying Atomicity Preservation and Deadlock Freedom of Generic Code

A challenging aspect of model-to-code transformations is to ensure that the semantic behavior of the input model is preserved in the output code. When constructing concurrent systems, this is mainly difficult due to the non-deterministic potential interaction between threads. In this chapter, we consider this issue for a framework that implements a transformation chain from models expressed in the state machine based domain-specific modeling language SLCO *to Java. In particular, we provide a fine-grained generic mechanism to preserve atomicity of* SLCO *statements in the Java implementation. We give its generic specification based on separation logic and verify it using the verification tool VeriFast. The solution can be regarded as a reusable module to safely implement atomic operations in concurrent systems. Moreover, we also prove with VeriFast that our mechanism does not introduce deadlocks. The specification formally ensures that the locks are not used reentrantly which simplifies the formal treatment of the Java locks.*

5.1 Introduction

Model transformation is a powerful concept in model-driven software engineering [70]. Starting with an initial model written in a domain-specific modeling language (DSML), other artifacts such as additional models, source code and test scripts can be produced via a chain of transformations. The initial model is typically written at a conveniently high level of abstraction, allowing the user to reason about complex system behavior in an intuitive way. The model transformations are supposed to preserve the correctness of the initial model, thereby realising a framework where the generated artifacts are correct by construction. A question that naturally arises for model-to-code transformations is how to guarantee that functional properties of the input models are preserved in the generated code [84]. In particular, this requires semantic conformance between the model and the generated code. For models in the area of safety-critical concurrent systems, the main complication to guarantee this equivalence involves the potential of threads to non-deterministically interact with each other.

Specifically, when variables are shared among multiple threads, the absence of race conditions is crucial to guarantee that no undesired updates of those variables can be performed. This relates to the notion of *atomicity* of the instructions executed by the threads. For instance, if two threads both increment the value of a variable x by one, then it is ensured that the final value of x equals the initial value plus two only when each of those increments can be performed atomically. Achieving atomicity of program instructions can be done using various techniques, such as locks, semaphores, mutexes, or CPU instructions such as *compare-and-swap*.

Also in modeling languages atomicity is an important concept, to simplify the reasoning about program instructions by abstracting away the atomicity implementation details. Hence, an important requirement for model-to-code transformations is that the atomicity of the statements in the modeling language is preserved in the code. A conceptual solution would be to map each statement to an atomic block in the implementation language. Strictly speaking, a block of instructions is atomic if during its execution no instruction of another thread is allowed to be executed. However, such a definition is too strong for practical purposes, since it excludes the possibility for threads to run truly concurrently in cases when they access different variables, and therefore do not interfere with each other. For this reason, it is usually replaced with weaker notions that still ensure non-interference. One such version, sometimes called *serializability* [21], allows instruction blocks to be executed concurrently as long as their individual results are not affected by the other blocks.

In this chapter, we demonstrate how one can establish that a model-to-code transformation transforms atomic statements in modeling languages to blocks of program instructions that are serializable. To illustrate this, we focus on the DSML presented in Chapter 2, i.e., Simple Language of Communicating Objects (SLCO), on the one hand, and the Java programming language on the other hand. It should be stressed, though, that our approach is suitable for any combination of a modern imperative programming language with concurrency and a modeling language that is, like SLCO, based on state machines that can be placed in parallel composition, and can change state by firing transitions with atomic statements (for instance, UML state machines).

The solution in this chapter, as a first step towards having completely verified generic code, addresses the challenge on the preservation of the atomicity property discussed in Chapter 3.

Contributions. First, we discuss how we have implemented, specified, and verified a protection mechanism to access shared variables in such a way that the code blocks implementing atomic DSML statements are guaranteed to be serializable. This generic mechanism is used in our framework to automatically transform SLCO models into multi-threaded Java code, but the solution is general enough to be used in other model-to-code transformations as well.

The mechanism employs a fine-grained ordered-locking approach. A coarse-grained approach tends to negatively impact the performance of multi-threaded software, while a lock-free approach, in particular using atomic instructions such as *compare-and-swap*, is necessarily restricted to work only for statements that involve only a single shared variable.

Second, we show the feasibility to verify the atomicity of generic statements by focusing on the SLCO assignment statement. We formally prove its implementation against a specification of non-interference using the VeriFast tool [68]. Being based on separation logic [86], VeriFast is suitable to deal with aliasing and concurrency in Java, as well

as with race conditions by using the concept of ownership of shared resources between multi-threaded programs.

Third, we introduce a wrapper class pattern to perform modular verification. With the wrapper class, it is possible to encapsulate data structures that are used in the code, but are not subjected to verification (for instance, because they have already been verified at an earlier stage possibly using a different tool).

In addition to the three contributions mentioned above, we also discuss how we can automatically prove that our mechanism to ensure the atomicity of statements does not introduce, what we call, *lock-deadlocks*. A lock-deadlock occurs when each thread in a set S is blocked trying to acquire a lock which is held by another thread in S. Since the methods involved in the mechanism are the only ones which manipulate locks, the mechanism being free of lock-deadlocks implies that our model-to-code transformation always produces programs that are lock-deadlock free. Lock-deadlock free is informally ensured by the assumption that the locks are always acquired in a certain fixed order. Using the implementation in VeriFast [64] of several modular verification techniques [64, 76], we are able to formally specify this requirement in the contracts of the relevant methods and to verify in a modular way that deadlocks are not introduced.

As an added value, we prove that in our generated programs there is no need for reentrant locks. This allows us to simplify the formal specification of the locks used in our mechanism. In [117], our verification already relied on this observation, but the current specification formally ensures adherence to it.

The remainder of the chapter is structured as follows. Section 5.2 describes the implementation of atomicity of SLCO statements in Java, as well as the implementation of the generic wrapper class. In Section 5.3, we demonstrate how to specify and verify the Java implementation with regard to the atomicity property. Section 5.4 contains the specification and verification of the deadlock and reentrance avoidance. Related work is discussed in Section 5.5, and Section 5.6 contains our conclusions and a discussion about future work.

5.2 Implementing SLCO Atomicity

In this section, we consider atomicity of SLCO statements; we provide a semantically comparable form of non-interference called serializability [21] for the Java blocks implementing those statements. Furthermore, to facilitate the transformation of SLCO statements to Java code, specifically to handle accessing shared variables, we introduce the generic data structure `SharedVariableList`.

In our model-to-code transformation, each SLCO statement s in a state machine M is transformed into a block of Java instructions $\sigma = \mathtt{s}_0; \mathtt{s}_1; \ldots; \mathtt{s}_n$ to be executed by a thread t_M. Strictly speaking, preserving atomicity of s in σ means that no instruction $\mathtt{s}'$ of any thread $t \neq t_M$ is allowed to be executed between the beginning and end of the execution of σ.

However, implementing atomicity in this strict sense is undesirable when constructing multi-threaded software, since it disallows true parallelism. That is why we replace this strong atomicity requirement with serializability. Serializability guarantees that for any concurrent execution of Java blocks (corresponding to an atomic SLCO statement) there exists a sequential execution of those blocks that is indistinguishable from the concurrent

execution, in terms of the final effect on the global system state. More explicitly, let σ and σ' be two different instruction blocks to be executed by different threads t_M, t'_M. Let q_0 be a global state in the Java model in which both σ and σ' can start a concurrent execution and let q_1 be a state in which the system ends up after the execution of both σ and σ'. Then, q_1 also is obtained after sequential execution of the sequence $\sigma\sigma'$ or $\sigma'\sigma$. Hence, we may reason about their execution as if σ was first completely executed before σ' was started, or vice versa. (Note that this also covers the case when σ may prevent the execution of σ' or vice versa.) The extension of the concept of serializability to an arbitrary number of instruction blocks σ_i is straightforward.

The state of a system is determined by the values of its variables. In SLCO, statements may access variables shared by multiple state machines. Therefore, in the corresponding Java code, multiple threads may access the same shared variables. In order to realize serializability in such a setting, it must be ensured that an instruction s' of some thread t cannot affect the variables accessed by the instructions in a block σ of thread $t_M \neq t$ running concurrently.

The way in which shared variables are protected has a significant impact on the overall performance of concurrent programs. For example, using one single global lock to protect a list frequently accessed by several threads is likely to scale much worse than when each element in that list is individually lockable.

SLCO statements may require access to just a subset of the shared variables of an object. Therefore, each element in the list of shared variables is assigned its own lock for read and write access. This gives a better performance than using a single lock for the complete list. In this way we achieve serializability, as shown in Section 5.3.2.

Individual locking may introduce deadlocks. We use the technique of ordered locking [57] to prevent them. The ordered locking mechanism guarantees that when multiple threads compete over a set of variables, one thread is always able to acquire access to all of them. Of course, other threads requiring access to different shared variables are able to access these concurrently.

Note that the locks can be released in an arbitrary order. Obviously there is no deadlock during the releasing since at least one method - namely, `unlockV` is active. After the locks are released we again have the situation in which multiple threads compete for the locks in a fixed order.

Our synchronization mechanism for shared variables is shown in Listing 5.1. The `SharedVariableList` at line 2, as a wrapper class, is introduced to abstract away how the list of shared variables is implemented. It can be used to encapsulate Java data structures. The methods `lockV` at lines 4-8 and `unlockV` at lines 9-13 are used to acquire and release each lock of the shared variables in the list.

Listing 5.1: Implementation of class `Statement`

```
1 public abstract class Statement {
2   protected SharedVariableList variablesList;
3   ...
4   public void lockV() {
5     for (int i = 0; i < variablesList.size(); i++) {
6       variablesList.get(i).lock.lock();
7     }
8   }
```

```
9    public void unlockV() {
10     for (int i = 0; i < variablesList.size(); i++) {
11       variablesList.get(i).lock.unlock();
12     }
13   }
14 }
```

As already mentioned, to store shared variables, class `SharedVariableList`, as a wrapper class, is introduced. Listing 5.2 shows its declaration. The concept of wrapper classes is quite common in object-oriented programming and is a pattern in object-oriented development, namely *Decorator* (also known as *Wrapper*) [105] . The wrapper class is used to hide information of concrete Java data structures, which allows modular implementation, specification and verification. Parts of the code that use `SharedVariableList` can be verified without involving the data structure; in fact, it may not even have been implemented yet. This helps to scale verification to larger programs, since the wrapper class needs to be analyzed only once, instead of once per call. Finally, modifying the implementation of the data structure encapsulated by the wrapper class never breaks correctness of its callers. This allows for simultaneous development and verification of code.

Listing 5.2: Declaration of class `SharedVariableList`

```
1 public final class SharedVariableList{
2   public SharedVariableList();
3   public int size();
4   public SharedVariable get(int index);
5   boolean add(SharedVariable e);
6 }
```

A `SharedVariableList` list contains all shared variables which are involved in the same SLCO statement. Each shared variable is an instance of the class `SharedVariable` which has three fields `name`, `value` and `lock`, as shown in Listing 5.3. The lock of each shared variable can be acquired and released via `lock()` and `unlock()` methods of Java class `Lock`, respectively.

Listing 5.3: Declaration of class `SharedVariable`

```
1 public class SharedVariable {
2   private String name;
3   private Object value;
4   protected Lock lock;
5   ...
6 }
```

The class `Assignment` as a subclass of class `Statement` (Listing 5.4) contains a method called `lockAndAssign` that can be used to safely (preserving serializability) assign a new value to a shared variable. The abstract method `assign` is implemented in the subclass which is related to a concrete SLCO assignment. When a thread attempts to execute the method `assign`, it will be delayed until all locks in the `variablesList` of variables to be accessed by the `assign` method are not being used anymore by other threads.

Listing 5.4: Implementation of class `Assignment`

```
1 public abstract class Assignment extends Statement {
2   ...
3   public abstract void assign();
4   public void lockAndAssign() {
5     lockV();
6     assign();
7     unlockV();
8   }
9 }
```

5.3 Specifying and Verifying SLCO Atomicity

In the previous section, we explained how the atomicity of SLCO statements can be implemented using serializability. We can use separation logic in VeriFast to verify the serializability, i.e., the fact that there is non-interference between different threads.

As described in Section 2.3, separation logic uses the principle of a minimal memory footprint, meaning that a separation assertion describes a unique heap. For example, the assertion $o.f \mapsto a * o.g \mapsto b$ describes a heap consisting of exactly two entries. This property together with the requirement that the heaps of two separate threads are disjoint, makes it possible to give a natural ownership interpretation of a shared resource. If a separation logic assertion P holds at some program location on a thread, we say that the thread owns the part of the heap described by P at that location.

When a thread acquires a shared object, it claims the ownership of the state associated with the variable; when releasing the variable, it must return the ownership of the corresponding piece of state. At all stages, our use of separation logic ensures that each piece of the heap is accessed by at most one thread. It thus becomes possible to reason about concurrent programs in which ownership of a shared variable can be perceived to transfer dynamically between threads. We achieve this dynamic transfer by associating invariants to locks of shared objects. The invariant representing the environment of the thread expresses the ownership of the shared variable.

By acquiring a lock, the verified program component also acquires the lock invariant representing the heap that corresponds to the shared variables. The invariant ensures a full permission to change the actual shared variables. By releasing the lock, the component releases, together with the invariant, also the acquired ownerships. This is expressed by the following rules for the *lock* and *unlock* operations

$$\{\mathbf{emp}\}\ \texttt{v.lock()}\ \{I_{\texttt{v}}(l)\} \tag{L}$$

$$\{I_{\texttt{v}}(l)\}\ \texttt{v.unlock()}\ \{\mathbf{emp}\} \tag{UL}$$

where $I_{\texttt{v}}(l)$ is the invariant associated with lock l of variable `v`. Note that this rule is only sound for non-reentrant locks.

The interpretation of correctness depends crucially on rules L and UL for the *lock* and *unlock* operations. In rules L and UL invariant $I_v(l)$ is of the form $v \mapsto \_$, i.e., expresses ownership of the variable v. By rule L, $I_v(l)$ is guaranteed to hold after `lock`, i.e., in the beginning of the protected code block. This means that the corresponding thread acquires

the ownership of v. Similarly, at the end of the block, after `unlock`, the ownership of v is released and the invariant $I_v(l)$ is no longer guaranteed to hold. Let V be the list of shared variables associated with the statement implemented in the block. By executing `lock` for each variable in V, using a combination of the L and UL rules and the frame rule from Section 2.3, an assertion I_V is established which is the conjunction of the invariants $I_v(l)$, for all $v \in V$. I_V can be seen as an invariant of the list V which expresses ownership of all variables in V by the thread. The concrete setting of our model transformation ensures that no shared variable in V is acquired or released within the protected code block. This is achieved by simply not using `lock` and `unlock` within the protected block, since these are the only statements with which one can acquire or release ownership. Together with the fact that invariant I_V holds at the beginning and at the end of the block, this implies non-interference since all variables are held exclusively by the thread during the execution of the block.

VeriFast supports modular verification in the sense that each method is verified separately. In this, each method relies on its environment to comply with the invariant. This is checked during the verification when several threads are combined. In the following sections, we specify and verify the atomicity of Java constructs corresponding with SLCO statements using separation logic via VeriFast.

5.3.1 Class SharedVariableList Specification

The wrapper class `SharedVariableList` is specified in separation logic in a way that is in fact independent of the Java programming language. In Listing 5.5, methods for modifying and querying instances of the class `SharedVariableList` are provided, such as `size`, `add` and `get`.

Listing 5.5: Specification of class `SharedVariableList`

```
1  final class SharedVariableList{
2    /*@ predicate List(list<SharedVariable> elements); @*/
3    public SharedVariableList();
4    //@ requires true;
5    //@ ensures List(nil);
6    public int size();
7    //@ requires [?f]List(?es);
8    //@ ensures [f]List(es) &*& result == length(es);
9    public SharedVariable get(int index);
10   //@ requires [?f]List(?es) &*& 0 <= index &*& index <
       length(es);
11   //@ ensures [f]List(es) &*& result == nth(index, es);
12   boolean add(SharedVariable e);
13   //@ requires List(?es);
14   //@ ensures List(append(es, cons(e, nil))) &*& result;
15 }
```

We express the state of the `sharedVariableList` instances using a mathematical list predefined in VeriFast as follows: The empty list is denoted by `nil` and a nonempty list starting by an element `h` and a tail `t` is denoted by `cons(h,t)`. In particular, predicate `List` is an abstract predicate that provides an abstract representation of the

contents of the list of shared variables. More concretely, parameter `elements` is a mathematical list containing the actual program variables that are stored in the list. The actual implementation can be, for instance, a dynamic list or an array. Using the abstract predicate we can hide such implementation details during the verification.

The pre- and postconditions form the *contracts* of the methods and they are denoted by the keywords `requires` and `ensures`, respectively, like in lines 4-5 in Listing 5.5. This contract of the constructor guarantees that an object is created corresponding to an empty list, regardless of the precondition.

The specification of `size` in lines 7 and 8 states that the method returns the length of the list. Component assertions of pre- and postconditions are separated by the spatial conjunction denoted by `&*&`. Both `[?f]` and `[f]` are fractional ownerships. The question mark `?` in front of a variable means that the matched value is bound to the variable and all later occurrences of that variable in the contract refer to this matched value. In our case the value of the fractional permission `f` in the precondition in line 7 must be the same as the one in the postcondition in line 8. Hence, the precondition in line 7 `[?f]List(?es)` expresses both that a fractional ownership with fraction `f` is required for the shared variable list corresponding with the mathematical list `es`, and records `f` and `es`. The postcondition specifies that the method returns the ownership of `es` to the caller with the same fraction `f` and the result that is returned by `size` is the length of `es`. `result` is a reserved variable name representing the return value of the method.

The precondition of `get` (line 10) requires that a valid element `index` is provided as an input parameter. The postcondition expresses that the element at position `index` in list `es` is returned using the mathematical function `nth`.

Unlike for other methods, the precondition of `add` (line 13) requires full ownership of the list `es`. As a result, the caller who owns `es` is allowed to insert an element into the list. Finally the method `add` returns the ownership of a new list that combines the list `es` with the newly inserted element `e` using the function `append`.

5.3.2 Class Statement Specification

The specification of class `Statement` is shown in Listing 5.6. The predicate constructor `lock_inv` in line 2 is essential for the preservation of serializability. It defines the lock invariant $I_{\mathrm{v}}(l)$ associated with the lock of a variable `v` passed as a parameter. The assertion `v.value |->_` asserts the full ownership of *v.value*. The underscore '_' denotes an arbitrary value.

The recursive predicates `locks` and `invariants` in lines 3-4 and 5-6, respectively, are used to specify dynamic data structures with unbounded-size like `Lists`. The recursive predicates are defined in VeriFast as ones that invoke themselves. More specifically, the body of each predicate is a conditional assertion. If `vs` is `null` (the base case of the induction) then the value of predicate `locks` is `true` (line 4); otherwise, the inductive step asserts that the lock of the first element of the list `vs`, `head(vs)` is partially owned. This is given by `return [_]head(vs).lock |-> ?lock`, where `[_]` denotes an unspecified fraction. Besides that, invariant `lock_inv(head(vs))` is associated with the lock of the first element, via the predicate `ReentrantLock`. Predicate `ReentrantLock` is defined by VeriFast as a specification of the `ReentrantLock` class, as shown in Listing 5.7. In a similar fashion, the recursive predicate `invariants` states that a conjunction of invariants corresponding to the (locks of the) variables in list `vs` is

recursively built. As mentioned above, for each variable the corresponding invariant is given by the specification `lock_inv(head(vs))`.

Predicate `Statement_lock()` is actually defined in Listing 5.8 and denotes that the `Statement` object is in a valid state corresponding to an abstract value given by the mathematical object list of shared variable objects `vs`. The body of method `lockV` needs to establish the above mentioned invariant I_V of each variable in list `vs`, which is expressed by the postcondition `invariants(vs)`. The postcondition `invariants(vs)` is also one part of the precondition in the method `unlockV`. By calling the method `unlockV`, invariant I_v of each variable in list `vs` is not guaranteed to hold anymore. After that, other threads can acquire the ownership of those variables through the method `lockV`.

Listing 5.6: Abstract specifications of class `Statement`

```
1 /*@
2 predicate_ctor lock_inv(SharedVariable v)(;) = v.value |-> _
    ;
3 predicate locks(list<SharedVariable> vs;) =
4   vs == nil ? true : [_]head(vs).lock |-> ?lock &*& [_]lock.
      ReentrantLock(lock_inv(head(vs))) &*& locks(tail(vs));
5 predicate invariants(list<SharedVariable> vs;) =
6   vs == nil ? true : lock_inv(head(vs))() &*& invariants(
      tail(vs));
7 @*/
8 class Statement {
9   //@ predicate Statement_lock(list<SharedVariable> vs);
10  void lockV();
11  //@ requires [_]Statement_lock(?vs);
12  //@ ensures invariants(vs);
13  void unlockV();
14  //@ requires [_]Statement_lock(?vs) &*& invariants(vs);
15  //@ ensures true;
16 }
```

The specification of class `ReentrantLock` provided by VeriFast is shown in Listing 5.7. The abstract predicate `ReentrantLock` at line 2 associates the invariant `inv` with the lock. The precondition for the lock creation (the constructor of class `ReentrantLock`), is that the invariant `inv` holds, as shown in line 5. The postcondition at line 6 ensures the created lock is associated with its invariant `inv`. The contract of the `lock` method is defined at lines 9-10. Predicate `ReentrantLock` in the precondition of method `lock` expresses the fact that the lock is available with an unspecified fraction. The postcondition of the method `lock` states that the invariant `inv` associated with the lock holds at this point. The specification of `unlock` is a mirror image of the contract of `lock` in the sense that basically the postcondition of the former is a precondition of the latter. By calling the method `unlock` at line 12, the invariant `inv` of a shared variable is not guaranteed to hold anymore. After that, other threads can acquire the ownership of this variable through the method `lock` at line 8.

Listing 5.7: Abstract specifications of class `ReentrantLock` to preserve atomicity

```
1 public class ReentrantLock{
```

```
2   //@ predicate ReentrantLock(predicate() inv);
3
4   public ReentrantLock()
5   //@ requires exists<predicate()>(?inv) &*& inv();
6   //@ ensures [_]ReentrantLock(inv);
7
8   public void lock();
9   //@ requires [_]ReentrantLock(?inv);
10  //@ ensures inv();
11
12  public void unlock();
13  //@ requires [_]ReentrantLock(?inv) &*& inv();
14  //@ ensures true;
15 }
```

5.3.3 Class Statement Verification

Above we gave the formal specification of the class `Statement` in an abstract, mathematically precise and implementation-independent way. To verify the implementation of class `Statement` against its specification, the specification of `SharedVariableList` in Listing 5.5 is a critical factor. Additional predicates and annotations are needed to verify the implementation of class `Statement`, as shown in Listing 5.8.

Listing 5.8: Verification annotations for class `Statement`

```
1 /*@
2 predicate_ctor lock_inv(SharedVariable v)(;) = ...
3 predicate locks(list<SharedVariable> vs;) = ...
4 predicate invariants(list<SharedVariable> vs;) = ...
5 @*/
6 class Statement {
7   SharedVariableList variablesList;
8   //@ predicate Statement_lock(list<SharedVariable> vs) =
       this.variablesList |-> ?a &*& a.List(vs) &*& locks(vs);
9   void lockV()
10  //@ requires [_]Statement_lock(?vs);
11  //@ ensures invariants(vs);
12  {
13    for (int i = 0; i < variablesList.size(); i++)
14    //@ requires [_]variablesList |-> ?b &*& [_]b.List(vs)
        &*& [_]locks(drop(i,vs)) &*& i >= 0 &*& i <= length(
        vs);
15    //@ ensures invariants(drop(old_i, vs));
16    {
17      //@ drop_n_plus_one(i,vs);
18      variablesList.get(i).lock.lock();
19    }
20  }
```

```
21   void unlockV()
22   //@ requires [_]Statement_lock(?vs) &*& invariants(vs);
23   //@ ensures true;
24   {
25     for (int  i = 0; i < variablesList.size(); i++)
26     //@ requires [_]variablesList |-> ?b &*& [_]b.List(vs)
          &*& [_]locks(drop(i,vs)) &*& invariants(drop(i,vs))
          &*& i >= 0 &*& i <= length(vs);
27     //@ ensures true;
28     {
29       //@ drop_n_plus_one(i,vs);
30       variablesList.get(i).lock.unlock();
31     }
32   }
33 }
```

The part in lines 1-5 in Listing 5.8 is identical to lines 1-7 in the specification Listing 5.6. Line 8 in Listing 5.8 contains the definition of predicate `Statement_lock` which in Listing 5.6 is only specified. The part `this.variablesList |-> ?a` states that field `variablesList` is defined. The last two conjuncts `a.List(vs) &*& locks(vs)` relate `variablesList` with the variable `vs` of type `list` and moreover the variables in `vs` are connected to their corresponding locks. The contract of method `lockV` in lines 10-11 is the same as the one in Listing 5.6.

To obtain the lock of each element in the `SharedVariableList`, we use a `for` loop in line 13. Besides loop invariants, VeriFast supports also loop verification by specifying a loop contract consisting of a precondition and a postcondition [100]. Then the loop is verified as if it were written using a local recursive function. The contract at lines 14-15 specifies the permissions used only by a specific recursive call (i.e., corresponding to a specific value of the loop counter `i`). More specifically, the precondition in line 14 matches `variablesList` with variable `b` (`[_]variablesList |-> ?b`), relates `b` to variable `vs` (`[_]b.List(vs)`), associates the variables from the `i`-th to the `vs.length-1`-th in list `vs` (`[_]locks(drop(i,vs))`) to their locks, and finally limits the range of the counter `i` (`i >= 0 &*& i <= length(vs)`). The segment `vs` from `i` to `vs.length-1` is obtained using the built-in function on lists `drop`. In a similar way in the postcondition in line 15, the list tail starting with `old_i` is obtained as an argument of the predicate `invariants`. Variable `old_i` refers to the value of the variable `i` at the start of the virtual function call (loop body). After the loop termination, `old_i` equals 0. This implies the validity of the conjunction of all lock invariants and consequently the ownership of all variables in `vs`.

Lemma functions are used to help VeriFast transform one assertion to another. The contract of a lemma function corresponds to a theorem, its body to a proof, and a call to an application of the theorem. Lemma function `drop_n_plus_one(i,vs)` in line 17 tells the verifier that `drop(n,vs)` is equivalent to the concatenation of the element `nth(n,vs)` with the list `drop(n+1,vs)`.

The detailed annotation of `unlockV` (lines 21-32) uses the same concepts as the annotation of `lockV`, therefore we do not discuss `unlockV` explicitly. It expresses that in each iteration the loop invariant shrinks instead of grows with the addition of a new

conjunct, i.e., invariant associated with a lock.

The specification and annotation in the current section is sufficient to prove that predicate `invariants`, which corresponds to I_V, holds at the beginning and at the end of the block implementing the statements. In Listing 5.4 this means that `invariants` holds immediately after `lockV` in line 5 and immediately before the `unlockV` in line 7. The validity of `invariants` ensures ownership of the variables in `sharedVariables`. As discussed above, by construction of our transformation (i.e., by not using `lock` and `unlock` within the protected block of the statement translation) we ensure that the block does not release this ownership. For example, in Listing 5.4 methods `assign` in line 6, as well as the implementations of all other types of SLCO statements, satisfies this property. VeriFast is able to verify that all relevant variables are held by the thread executing the method corresponding to the implementation of the SLCO statement. This implies serializability of the programs as can be seen from the following arguments.

Consider two instruction blocks σ and σ' which both implement an SLCO statement. Hence, they both contain a lock protected code block. We show that they are serializable. Let V and V' be the set of variables accessed by σ and σ', respectively. Consider first the case when $V \cap V' \neq \emptyset$. Suppose that σ first acquires the ownership of all variables in V. Then σ' must wait until those variables are released. If there is some prefix σ'' of σ' which has been executed before σ acquired the variables in V, then σ' could have modified only variables which are not in V. So, this prefix could have been executed after σ terminated and therefore the sequence $\sigma\sigma'$ will produce the same variable changes, i.e., the same state as the concurrent execution of σ and σ'.

A similar argument can be used for the case $V \cap V' = \emptyset$. In this case the individual statements in σ and σ' are independent of one another and can be permuted in an arbitrary order. The set of possible sequences includes both $\sigma\sigma'$ and $\sigma'\sigma$ and they confluently lead to the same end state.

5.4 Specifying and Verifying Lock-Deadlock Freedom

In this section, we show how in addition to specifying and verifying atomicity preservation, we can also verify that programs generated by our model-to-code transformation are guaranteed to be free of lock-deadlocks. We first discuss the theory behind the approach and after that we present the concrete specification and verification in VeriFast.

5.4.1 Lock-deadlock Freedom for Generated Code

Code generation from models should preserve deadlock freedom: if the model is deadlock free, the generated code should be deadlock free. This places demands on the generic code and on the translation.

Here, we consider an important category of deadlocks: *lock-deadlock* - a set of threads is lock-deadlocked if all threads in the set are blocked because they try to acquire locks that are already acquired by some thread in the set [64, 76].

In the spirit of model-driven development we treat lock-deadlock freedom as a property to be ensured by the code generation rather than having to establish this for each specific instance of generated code.

A well-established way to obtain lock-deadlock freedom is putting an ordering on

acquiring locks. To achieve lock ordering at the level of code generation we put requirements on the order of acquiring locks by means of the specification of the lock, the lock API. This specification is such that any code (generated or obtained otherwise) that satisfies the precondition of the lock method of this API is lock-deadlock free. (Note that here the approach is SLCO independent and not Java specific.) We show how these requirements are met by the generic code and the way it is used in the translation from SLCO to Java.

Conditions for Lock-deadlock Freedom Let the locks be ordered by an anti-reflexive partial order $<$. In the API specification of the lock, the auxiliary variable *lockset*(T) represents the locks acquired by thread T, which is maintained by the methods lock() and unlock(). The following assumption is a cornerstone of the lock-deadlock avoidance approach:

Assumption 1 (Lock Acquire Precondition) . The precondition of method `lock` of lock *lck* which is to be acquired implies that $lck < l$ for any l in *lockset*(T), where T is the current thread.

Theorem 1 (Lock-deadlock freedom)**.** *Code adhering to the precondition of method* `lock` *as in Assumption 1 is lock-deadlock free.*

Proof. Suppose, by contradiction, that we have a lock-deadlock set S. Note that S is finite and non-empty. Take a thread T_1 from S. Then T_1 is waiting for, say, lock l_1. Since S is a lock-deadlock set, there is a thread T_2 in S holding l_1. T_2 is in S, so it is also waiting for a lock, say l_2. Since T_2 has tried to acquire l_2 by a call to the method `lock`, the precondition applies (Assumption 1) and $l_1 > l_2$. Suppose T_3 in S is holding l_2, and it is also waiting for a lock, say l_3. With the same reasoning we find $l_2 > l_3$. Continuing this we get an infinite chain $l_1 > l_2 > \cdots > l_n > \ldots$ All l_i are different, so we have an infinite set of locks. As every lock in the chain has a thread in S that is waiting for it and every thread is waiting for one lock, hence S is infinite. This is in contradiction with the fact that S is finite.

□

Lock-deadlock Freedom for Java Code Generated from SLCO Models We show that code as generated from SLCO specifications adheres to the precondition of method `lock` as in Assumption 1 and hence is, by Theorem 1, lock-deadlock free.

In the following Lemma 2 we need the formal specification of method `lockV` of class `Statement` in Listing 5.10.

Lemma 2. *If the precondition of* `lockV` *of class* `Statement` *in Listing 5.10 is satisfied, the generic code adheres to the precondition of method* `lock` *as in Assumption 1.*

Proof. (sketch) The only place where `lock` is called in the generic code is in the method `lockV`. There `lock` methods are called in the designated order (since the precondition implies the correct ordering of variables/locks) and hence the locks will be acquired in the same correct descending order. The only possible problem is the first lock to be acquired by `lockV`, if it is greater than some of the locks in *lockset*(T). As the only place where `lock` is called in the generic code is in the method `lockV` and the precondition of `lockV`

requires that all locks that are acquired by `lockV` are below all locks in `lockV`, the problem can be avoided. □

In Section 5.4.2, using VeriFast we formally prove that the implementation of method `lockV` of class `Statement` satisfies its specification in Listing 5.10.

Lemma 3. *The generated code adheres to the specification of class `Statement`.*

Proof. Inspecting the translator shows that the methods of instance s of class `Statement` are called only in the template method `lockAndAssign` of class `Assignment` as shown in Listing 5.4. We give a pseudo-formal annotation, where V is the set of locks that correspond to the variables of s.

$$\{\ \mathit{lockset}(T) == \emptyset\ \}$$
`lockV();`
$$\{\ \mathit{lockset}(T) == V\ \}$$
`assign();`
$$\{\ \mathit{lockset}(T) == V\ \}$$
`unlockV();`
$$\{\ \mathit{lockset}(T) == \emptyset\ \}$$

Method `assign` is implemented in the translation according to the specific assignment statement in the SLCO model. Inspecting the translator shows that it will not call `lock`, hence keeping the lockset invariant. To satisfy the above annotation it is needed that the precondition of `assign` is implied by the postcondition of `lockV`, and that the postcondition of `assign` implies the precondition of `unlockV`, which can be shown straightforwardly by inspecting the code of the translator. □

Lemmas 3 and 2 and Theorem 1 imply lock-deadlock freedom immediately for the generated code.

5.4.2 Formal Specification and Verification of Lock-Deadlock Freedom in VeriFast

To implement the theoretical considerations from the previous section we use the feature of VeriFast that ensures lock-deadlock freedom based on ordered lock acquisition [64]. However, the VeriFast specification of this feature is only defined for a lock struct in C. To this end we first need to adapt this specification to the one for locks in Java.

5.4.2.1 Class ReentrantLock Specification

Besides the freedom of deadlocks, the specification for locks in Java also needs to guarantee absence of lock reentrance. This is because the locks in our mechanism are never reacquired, i.e., a thread never attempts to acquire the lock again while holding it. The requirement for absence of lock reentrance also simplifies the formal specification of Java class `ReentrantLock`. The specification of this class is given in Listing 5.9.

Listing 5.9: Abstract specifications of class `ReentrantLock` to avoid deadlock

```
1 //@ predicate lockset(int threadId, list<ReentrantLock>
    lockIds);
```

```
2 public class ReentrantLock  {
3   //@ predicate ReentrantLock(predicate() inv);
4   //@ predicate ReentrantLocked(predicate() inv, int
      threadId, real frac);
5   public ReentrantLock()
6   //@ requires create_lock_ghost_args(?inv, ?ls, ?us) &*&
      inv() &*& lock_all_below_all(ls, us) == true;
7   //@ ensures [_]ReentrantLock(inv) &*& lock_above_all(this,
       ls) == true &*& lock_below_all(this, us) == true;
8
9   public void lock();
10  //@ requires [?f]ReentrantLock(?inv) &*& lockset(
      currentThread, ?locks) &*& lock_below_top(this, locks)
      == true;
11  //@ ensures ReentrantLocked(inv, currentThread, f) &*& inv
      () &*& lockset(currentThread, cons(this, locks));
12
13  public void unlock();
14  //@ requires ReentrantLocked(?inv, currentThread, ?f) &*&
      inv() &*& lockset(currentThread, ?locks);
15  //@ ensures [f]ReentrantLock(inv) &*& lockset(
      currentThread, remove(this, locks));
16 }
```

The abstract predicate `lockset` in line 1 is defined by the client that uses the specification. It states that the `ReentrantLock` list `lockIds` contains the locks held by thread `threadId`. The abstract predicate `ReentrantLock` associates the invariant `inv` with the lock. Predicate `ReentrantLocked` denotes that the lock is associated with invariant `inv` and it is held (owned) by `threadId` with fraction `f`.

The contract of the constructor `ReentrantLock` at lines 6-7 associates the created lock with its invariant `inv` and places the lock to be created in the partial ordering above the locks in set `ls`, i.e., the lower bound of the lock, and below the locks in set `us`, i.e., the upper bound of the lock. The precondition for the lock creation requires that `inv` holds and also that the lower and upper bounds of the lock are consistent, i.e., that the locks in `ls` are all below all locks in `us`, as shown at line 6.

As discussed in Section 5.4.1, the assumption to acquire a lock *lck* for a thread T is that $lck < l$ for any lock l in $lockset(T)$, where $lockset(T)$ is a set of locks held by T. Therefore, the precondition of method `lock` at line 10 needs to express the decreasing order of lock acquisition, i.e., the requirement that the lock needs to be below all locks currently held by the current thread. This can be guaranteed by using the fixpoint function `lock_below_top(this, locks)` in the precondition. Variable `currentThread` in line 10 is a built-in variable in VeriFast to represent the current thread. Note that if the lock is reacquired, it is already held by the current thread, i.e., included in `locks`, therefore an error will be signaled by VeriFast. Hence, the specification also implies absence of reentrance. Predicate `ReentrantLock` in the precondition expresses the fact that the lock is available with fraction `f`. The postcondition states that the lock is owned (locked) with fraction `f`, that the associated invariant holds at this point and that the lock is added

to the `lockset` held by the current thread.

The specification of `unlock` is a mirror image of the contract of `lock` in the sense that basically the postcondition of the former is a precondition of the latter and vice versa.

5.4.2.2 Class Statement Specification

The updated specification of class `Statement` is shown in Listing 5.10. Predicates `lock_inv` in line 2 and `invariants` in lines 5-6 are the same as in Listing 5.6. The recursive predicate `locks` in lines 3-4 has an output parameter called `ll` that is a `ReentrantLock` list associated with the `SharedVariable` list `vs`. The body of `locks` is a conditional assertion. If `vs` is `null` then `ll` is `null` (line 4) too; otherwise, the inductive step asserts that the lock of the first element of the list `vs`, i.e., `head(vs)`, is added at the head of `ll` denoted by `ll == cons(lock, ll0)`. Besides that, `lock_above_all` is a fixpoint function defined by VeriFast to ensure the level of `lock` is above the level of each element in list `ll0` which is the tail of `ll`. In a similar fashion, the recursive predicate `locked` in lines 7-8 states that invariant `lock_inv(head(vs))` is associated with the lock of the first element, via the predicate `ReentrantLocked`. Predicate `ReentrantLocked` also states that thread `t` is the current thread holding the lock of the first element. The semicolon (`;`) in line 7 in the parameter list of predicate `locked` is used to declare `locked` as a *precise* predicate [68], which enables VeriFast's logic for automatically opening and closing this predicate.

The lemma function `extend_upper_bound_at_top` in lines 9-14 states that if the lock list `ys` is an upper bound of list `xs`, i.e., the level of each lock in list `xs` is below the level of each lock in list `ys`, and the level of lock `x` is above the level of locks in list `xs`, then we can add `x` at the top of `ys` and the new list `cons(x,ys)` will remain an upper bound of `xs`. The `lemma_auto` function is another type of lemma which can be implicitly called by VeriFast. `locks_inv()` in lines 15-20 instructs VeriFast to always replace the chunk `[?f]locks(?vs, ?ls)` with `[f]locks(vs, ls) &*& vs != nil || ls == nil`. (The lemma is proved by opening the predicate `locks` explicitly.)

Predicate `Statement_lock()` in line 23 is similar to the one defined in Listing 5.6 but has two parameters in its parameter list. The list of `ReentrantLock` objects `ll` is used to extract the list of locks associated with shared variable objects `vs`.

The precondition of `lockV` referred to in Lemma 2 is given in line 25. The predicate `lockset` states that the list `levelList` is a list of locks acquired so far by the current thread. Predicate `lock_all_below_all` requires that levels of the locks to be acquired in list `ll` are below the levels of the locks in `levelList`, i.e., the locks acquired so far by the current thread.

The postcondition of method `lockV` implies the correctness of the annotation in the proof of Lemma 3. It asserts that all variables in `vs` are locked by the current thread and that consequently each element in list `ll` is added to the list `levelList`. VeriFast requires that the order of lock acquisition in list `levelList` is descending. However, EGL we used for defining the SLCO-to-Java transformation only supports a method that sorts the locks in the list `ll` into ascending order. Therefore, the order of elements in list `ll` should be reversed before appending the list `ll` to the list `levelList`. This is expressed by the predicate `lockset(currentThread,append(reverse(ll),levelList))` in line 26. Actually our transformation ensures that the list of acquired locks `levelList` is always empty at the precondition in line 25. So, the requirement that all locks in `ll` are

below all locks in `levelList` is trivially satisfied.

The predicates `lockset(currentThread,?levelList)` and `invariants(vs)` in the postcondition of the method `lockV` are also part of the precondition in the method `unlockV`. By calling the method `unlockV`, invariant I_v of each variable in list `vs` is not guaranteed to hold anymore and all locks in `ll` are also removed from the list `levelList` as expressed by `remove_all(ll,levelList)`.

Listing 5.10: Abstract specifications of class `Statement` to avoid deadlock

```
1  /*@
2  predicate_ctor lock_inv(SharedVariable v)(;) = v.value |-> _
       ;
3  predicate locks(list<SharedVariable> vs; list<ReentrantLock>
       ll) =
4    vs == nil ? ll == nil : [_]head(vs).lock |-> ?lock &*& [_]
       lock.ReentrantLock(lock_inv(head(vs))) &*& locks(tail(
       vs), ?ll0) &*& ll == cons(lock,ll0) &*& lock_above_all(
       lock,ll0) == true;
5  predicate invariants(list<SharedVariable> vs;) =
6    vs == nil ? true : lock_inv(head(vs))() &*& invariants(
       tail(vs));
7  predicate locked(list<SharedVariable> vs, int t;) =
8    vs == nil ? true : [_]head(vs).lock |-> ?lock &*& lock.
       ReentrantLocked(lock_inv(head(vs)), t, _) &*& locked(
       tail(vs),t);
9  lemma void extend_upper_bound_at_top(ReentrantLock x, list<
       ReentrantLock> xs, list<ReentrantLock> ys)
10 requires lock_all_below_all(xs, ys) == true &*&
       lock_above_all(x,xs)== true;
11 ensures lock_all_below_all(xs, cons(x, ys)) == true;
12 {
13   switch (xs) { case nil: case cons(h, t):
       extend_upper_bound_at_top(x, t, ys); }
14 }
15 lemma_auto void locks_inv()
16 requires [?f]locks(?vs, ?ls);
17 ensures [f]locks(vs, ls) &*& vs != nil || ls == nil;
18 {
19   open locks(vs, ls);
20 }
21 @*/
22 class Statement {
23   //@ predicate Statement_lock(list<SharedVariable> vs, list
       <ReentrantLock> ll);
24   void lockV();
25   //@ requires [_]Statement_lock(?vs,?ll) &*& lockset(
       currentThread,?levelList) &*& lock_all_below_all(ll,
       levelsList) == true;
```

```
26   //@ ensures invariants(vs) &*& lockset(currentThread,
        append(reverse(ll),levelList)) &*& locked(vs,
        currentThread);
27
28   void unlockV();
29   //@ requires [_]Statement_lock(?vs,?ll) &*& invariants(vs)
         &*& lockset(currentThread,?levelList) &*& locked(vs,
        currentThread);
30   //@ ensures lockset(currentThread, remove_all(ll,levelList
        ));
31 }
```

5.4.2.3 Class Statement Verification

Above we provide the formal specification of the class `Statement` to avoid lock-deadlock in an abstract and modular way. Here, we verify the implementation of this class, as shown in Listing 5.11.

The definition of predicates in lines 2-7 in Listing 5.11 can be found in Listing 5.10 in lines 2-20. Line 11 in Listing 5.11 shows the definition of predicate `Statement_lock` in line 23 in Listing 5.10. It states that the field `variablesList` of class `Statement` is defined (`this.variablesList |-> ?a`) and it is related with the mathematical variable `vs` of type `list` (`a.List(vs)`). Moreover, the variables in `vs` are connected to their corresponding locks which are stored in the list `ll` (`locks(vs, ll)`). The precondition and postcondition of method `lockV` in lines 13-14 are the same as the ones in lines 15-26 in Listing 5.10.

Similar to the contract of the `for` loop for proving atomicity in Listing 5.8, we introduce the level of locks associated with `variablesList` into the contract of the `for` loop in Listing 5.11 to prove lock-deadlock freedom. In the precondition, the conjunct `[_]locks(drop(i,vs),?ll1)` associates the variables in `vs` from index `i` to `vs.length-1` with their locks and stores them in the list `ll1`. In a similar way in the postcondition in line 18, the list tail starting with `old_i` is obtained as an argument of the predicates `invariants` and `locked`. The predicate `invariants` in the postcondition ensures the ownership of all variables in `vs`. The predicate `locked` in the postcondition implies the validity of the postcondition of each `lock.lock()` of all locks associated with `vs`. After the execution of the `for` loop, `levelList1` is updated by appending the lock list `ll1` to it. To ensure that the locks in list `levelList1` are placed in descending order, the order of list `ll1` is reversed before appending it to the list `levelList1` via `reverse(ll1)`.

The `switch` statement in line 21 helps VeriFast to access the list `levelList1` even if it is `null`. The lemma function `extend_upper_bound_at_top` in line 23 works as described in Listing 5.10.

Another lemma function provided by VeriFast `append_assoc()` states the associative property of the `append` operator, which can append one list to another.

Instead of using `true` as the postcondition of method `unlockV`, the predicate `lockset(currentThread,remove_all(ll1,levelList1))` is used here for proving the atomicity property. It expresses that the locks of the variables in the list `ll1` are removed from the list `levelList1` after the execution of method `unlockV`. In each

iteration of the `for` loop, the element that is equal to the head of list `ll1` is removed from `levelList1`. The lemma function `remove_all_head(ll1,levelList1)` tells VeriFast that `remove_all(tail(ll1),remove(head(ll1),levelList1))` is equivalent to `remove_all(ll1,levelList1)`. The fixpoint function `remove(head(ll1), levelList1)` removes the element that is equal to the head of `ll1` from `levelList1` and `remove_all(ll1,levelList1)` removes all elements that occur in list `ll1` from `levelList1`.

Listing 5.11: Verification annotations for class `Statement` to avoid deadlock

```
1 /*@
2 predicate_ctor lock_inv(SharedVariable v)(;) = ...
3 predicate locks(list<SharedVariable> vs;) = ...
4 predicate invariants(list<SharedVariable> vs;) = ...
5 predicate locked(list<SharedVariable> vs, int t;) = ...
6 lemma void extend_upper_bound_at_top(ReentrantLock x, list<
    ReentrantLock> xs, list<ReentrantLock> ys)...
7 lemma_auto void locks_inv()...
8 @*/
9 class Statement {
10   SharedVariableList variablesList;
11   //@ predicate Statement_lock(list<SharedVariable> vs,list<
       ReentrantLock> ll) = this.variablesList |-> ?a &*& a !=
        null  &*& a.List(vs) &*& locks(vs,ll);
12   void lockV()
13   //@ requires [_]Statement_lock(?vs,?ll) &*& lockset(
       currentThread,?levelsList) &*& lock_all_below_all(ll,
       levelsList) == true;
14   //@ ensures invariants(vs) &*& lockset(currentThread,
       append(reverse(ll),levelsList)) &*& locked(vs,
       currentThread);
15   {
16     for (int i = 0; i < variablesList.size(); i++)
17     //@ requires [_]variablesList |-> ?b &*& [_]b.List(vs)
         &*& [_]locks(drop(i,vs),?ll1) &*& i >= 0 &*& i <=
         length(vs)&*& lockset(currentThread,?levelsList1) &*&
          lock_all_below_all(ll1,levelsList1) == true;
18     //@ ensures invariants(drop(old_i, vs)) &*& lockset(
         currentThread,append(reverse(ll1),levelsList1)) &*&
         locked(drop(old_i, vs), currentThread);
19     {
20       //@ drop_n_plus_one(i,vs);
21       //@ switch (levelsList1) { case nil: case cons(h,t): }
22       //@open(locks(_,_));
23       //@extend_upper_bound_at_top(head(ll1), tail(ll1),
           levelsList1);
24       //@append_assoc(reverse(tail(ll1)), {head(ll1)},
           levelsList1);
```

```
25        variablesList.get(i).lock.lock();
26      }
27    }
28
29    void unlockV()
30    //@ requires [_]Statement_lock(?vs,?ll) &*& invariants(vs)
          &*& lockset(currentThread,?levelList) &*& locked(vs,
         currentThread);
31    //@ ensures lockset(currentThread, remove_all(ll,levelList
         ));
32    {
33      for (int  i = 0; i < variablesList.size(); i++)
34      //@requires [_]variablesList |-> ?b &*& [_]b.List(vs)
           &*& [_]locks(drop(i,vs),?ll1) &*& invariants(drop(i,
           vs)) &*& i >= 0 &*& i <= length(vs) &*& lockset(
           currentThread,?levelList1) &*& locked(drop(i,vs),
           currentThread);
35      //@ ensures lockset(currentThread,remove_all(ll1,
           levelList1));
36      {
37        //@ drop_n_plus_one(i,vs);
38        //@ open(locked(_,_));
39        //@ open(locks(_,_));
40        //@ remove_all_head(ll1, levelList1);
41        variablesList.get(i).lock.unlock();
42      }
43    }
44  }
```

5.5 Related Work

The detection of race condition violations in concurrent code using the lock mechanism has been addressed by a number of type-based [43], static [9,42] and dynamic analysis [31] tools. However, as shown in [48], even code free of race conditions may still contain errors caused by intricate interaction of threads working on shared objects. Therefore, stronger concepts of non-interference, i.e., atomicity, are needed. In [48], atomicity is formally defined and an atomic type system was implemented to check it. The tool DoubleChecker [21] checks for serializability of concurrent programs based on run-time information about the dependences between threads. The above mentioned works check the correctness of programs a posteriori, i.e., after they have been fully implemented. In contrast, our approach statically verifies generic code to be used in the construction of complete programs, which improve re-usability and scalability of the verification.

5.6 Conclusions and Future Work

We have presented an approach for the verification of atomicity preservation in model-to-code transformations based on separation logic using the tool VeriFast. We applied this approach in the transformation from the domain-specific modeling language SLCO to Java.

To improve performance, we replaced the strong atomicity requirement of SLCO with the semantically relaxed notion of serializability. This was implemented by a fine-grained deadlock-free ordered locking mechanism allowing true parallelism. We stated the serializability in terms of ownership of shared variables expressed by means of lock invariants. Using VeriFast we verified non-interference in the Java code.

A strong aspect of our approach is that we can also formally prove that our mechanism does not introduce so-called lock-deadlocks caused by mutual blocking of threads waiting to acquire locks. We can do this in an automatic and modular fashion using VeriFast. The same specification for showing absence of lock-deadlocks allows us to prove that the locks are not reentrant. This simplifies the specification and formal reasoning.

Besides shared variables, SLCO also allows the use of channels for communication. The communication channel is extracted as a model independent concept in our framework. In the next chapter, we present a simple implementation of the SLCO asynchronous channel as well as its verification.

As a next step, an interesting direction is to verify that the generic code involving both locks and message-passing (channels) are free of deadlocks. Another valuable direction for future work is to address the verification of model-specific code. This would allow us to conclude that our transformation is guaranteed to produce correct code for any given correct SLCO model.

Chapter 6

Modular Verification of SLCO Communication Channels

Our approach to setting up the model-to-code transformation step is to distinguish between model-generic parts and model-specific parts of SLCO *models, as discussed in Chapter 3. An example of a model-generic* SLCO *concept is the communication channel. In this chapter, we demonstrate a simple implementation of* SLCO *asynchronous channels and also verify the implementation in an modular way. In particular, we focus on how to formally specify the behavior of the communication channel, such that modular verification of code using such model independent concepts is possible. To this end, we use a parameterized modular approach; we apply a novel proof schema that supports fine grained concurrency and procedure-modularity, and use the separation logic based tool VeriFast. Our results show that such tool-assisted formal verification can be a viable addition to traditional techniques, supporting object orientation, concurrency via threads, and parameterized verification.*

6.1 Introduction

Our approach to implement and verify model-to-code transformations relies on identifying the model-generic parts, as mentioned in Chapter 3. The model-generic concepts can be transformed to Java once, and from that moment on can be referred to in all code generated from specific SLCO models. An example of a model-generic aspect of SLCO models is the fine-grained mechanism for preserving atomicity of SLCO statements in Java implementations. This mechanism is generic in the way that it is applicable to all SLCO statements. In Chapter 5, we provided its generic specification based on separation logic and verified it using the verification tool VeriFast. In this chapter, we investigate another example of model-generic parts of SLCO models, i.e., the communication channel. This concept is generic in the sense that it is reused in the translation of all specific SLCO models consisting of objects that communicate with each other through channels.

As described in Chapter 2, SLCO channels are used for communication between SLCO objects. In particular, each object is described in terms of a finite number of concurrently operating state machines that can share variables. State machines in different objects

communicate with each other through channels. After a chain of transformations of SLCO models [41], in which incrementally more concrete information about the specified system can be added, multi-threaded Java code can be generated based on the last SLCO model as shown in Chapter 4. There each SLCO state machine is mapped to its own thread. Channels can be considered as shared data structures between concurrent threads in the Java implementation.

SLCO has a coarse granularity that supports reasoning about concurrency at a convenient high level of abstraction. On the other hand, the generated Java code implements concurrency through multi-threading, with a level of granularity that is much more fine-grained. For instance, synchronization of operations on channels shared between concurrent threads can be achieved via fine-grained internal mechanisms instead of coarse-grained external ones. This means that each operation acquires and releases access to multiple critical regions (CR) during its execution, instead of following a coarse-grained approach in which the complete operation is executed in one big critical region. This approach can offer better performance [67], which facilitates the development of correct, well-performing, complex software.

However, the code generation step is challenging to implement, since the transformation from coarse to fine-grained concurrency needs to be done in a way that correct and well-performing software is generated. Moreover, our goal is to verify the framework from SLCO models to Java code in a modular way, which in turn requires modularly specifying channel operations that perform fine-grained internal synchronization. In particular, we focus on *procedure-modularity* [67], i.e, each group of procedures (or operations) that cooperate to implement the SLCO channel construct can be verified separately, again under a well-defined, concise set of assumptions on its environment that performs proper abstraction over implementation aspects. However, fully specifying operations that perform fine-grained internal synchronization in a modular way is still a hard and open problem [67]. Furthermore, proving correctness of a framework that produces programs from models is fundamentally more complex than verifying an individual program [84].

In view of these observations, we distinguish the model-generic part and the model-specific part of asynchronous channels for SLCO models and then present the model-generic part as one part of generic code in the framework that supports the transformation from SLCO models to Java code. By making the distinction, the implementation of the model-generic part of SLCO channels can be updated without affecting the overall transformation machinery. This makes it possible to have different implementations of SLCO asynchronous channels, as discussed in Chapter 3. In addition to this, the complexity of proving the correctness of a channel implementation can be lowered. Generic code for implementing model-generic parts of SLCO channels can largely be treated as any other program, apart from the fact that it raises new concerns regarding the larger program context in which code constructs can be placed; these concerns are covered in this chapter. As a result, the remaining proof obligations for the transformation as a whole can be simplified; once we turn our attention to the specific code, we can directly use the specifications of the generic code constructs.

Compared to the implementation of SLCO asynchronous channels presented in Chapter 4, the one presented in this chapter is simplified as it does not cover all features. For instance, conditional receiving operations are not supported. The reason for this is to lower the complexity of fully specifying and verifying the channel operations that perform fine-grained internal synchronization in a modular way.

Additionally, we introduce a new modular specification schema to specify the behavior of modeling constructs in a setting where 1) fine-grained parallelism is used, and 2) the environment is general, i.e., we do not need to know anything about the environment to specify the constructs. Compared to [67], the schema presented in this chapter allows a better abstraction from the implementation details of the methods being specified.

Finally, we demonstrate our approach by specifying and verifying the presented implementation of SLCO asynchronous channels. This shows the feasibility of the approach, but also that judicious choices of implementation language, specification language, verification approach and tooling are required.

The remainder of this chapter is structured as follows. In Section 6.2, the new modular specification schema is described. In Section 6.3 we present the simplified implementation of SLCO asynchronous channels and in Section 6.4 we demonstrate how to apply the schema to specify and verify this simplified implementation, using VeriFast. Section 6.5 discusses related work. Conclusions and directions for further research are given in Section 6.6.

6.2 The Modular Specification Schema

Our aim is to specify modeling constructs and verify the implementation of those constructs in a modular way, meaning that each construct and its implementation should be independently specifiable and verifiable. The benefits of a modular approach are 1) that it will scale better than a monolithic approach and 2) that once a construct has been specified, we can abstract away its implementation details when verifying properties of the system.

Each construct implemented in a modular way as a procedure needs to be verified separately under a well-defined, concise set of assumptions on its environment that performs proper abstraction over implementation aspects. Because of this, and the fine-grained nature of the generic code, standard methods like the resource-invariant-based (RI) method [82] proposed by Owicki and Gries do not suffice. The reason for this is that the RI method in [82] requires auxiliary variable annotations that break the modularity of each construct. In [67], a modular specification schema was proposed to solve this problem. In this section, we introduce an improved version of this modular approach which, compared to [67], provides a better abstraction from the implementation of the verified method.

6.2.1 Verifying Methods with Fine-grained Parallelism

As already mentioned, the Java methods in our transformation framework implement fine-grained parallelism via fine-grained internal synchronization. This means that each method may acquire and release access to multiple CRs during its execution, instead of following a coarse-grained approach in which the complete method is executed in one big CR. As CRs tend to create performance bottlenecks in software, using multiple CRs decreases the level of dependency between threads in a multi-threaded system, and thereby increases the overall performance.

In order to verify methods with fine-grained parallelism, so-called *ghost code* must be inserted as part of the annotations. To see how this mechanism of code insertion works, we consider a method m belonging to a class `C` instantiated in an object `o`. We

want to give a specification of m in the form of a standard Hoare logic triple $\{P\}\mathtt{o}.m\{Q\}$. Under fine-grained parallelism one cannot formulate P and Q in terms of the actual fields determining the state of `o`. For instance, consider method `send(msg,G)` that sends a message `msg` to a channel queue `q` (`q` is a field of `C`), as in Listing 6.1. At line 8, the piece of code `G` given as a parameter to `send` is inserted.

Listing 6.1: A fine-grained `send` operation

```
1  class C
2    queue q
3    semaphore s
4    method send(msg,G)
5    begin
6      s.acquire()
7      q := q + msg
8      G
9      s.release()
10   end
```

In a concurrent setting, multiple threads may send messages to the queue of a single instance of `C` like in the example in Listing 6.2. The `||` at line 2 is used as the parallel execution operator between threads. We cannot write a specification of method `send` in terms of field `q`, e.g., we cannot claim that once the call of `send(a)` at line 2 is finished, the new content of `q` is `q+a`, where + indicates concatenation. This is because `q` may be changed by the call to `send(b)` between the execution of `send(a)` at lines 9 (semaphore release) and 10 (returning the control to the calling client program). This is analogous to Owicki-Gries, where global variables altered by multiple modules cannot be used directly to specify a module.[1]

To resolve this, ghost variables (also called logical or auxiliary variables) are added to the program. Ghost variables are write-only, i.e., the instrumented program can change them, but not read them. Hence, they do not change the control flow of the program and are only auxiliary verification devices. Each ghost variable is owned by a particular process, and only this process can potentially change its content. To illustrate the use of ghost variables, let us assume that `send` is used by a client program as shown in Listing 6.2.

Listing 6.2: A client using the `send` method.

```
1 o := New C()
2 o.send(a) || o.send(b)
```

Suppose we want to prove that if in the beginning of the program $len(\mathtt{q}) = 0$ holds, where *len* gives the length of the queue, then at the end, $len(\mathtt{q}) = 2$. We specify the two instances of `send` by introducing ghost variables y and z to capture the local effect on the length of `q` in the left and right method call, respectively. Resource invariant $I_A \equiv len(\mathtt{q}) = y + z$ captures how these local effects relate to the global resource (A is a tag which is associated with the resource [67]). Now we can specify `send(a)` with $\{y = 0\}\mathtt{send(a)}\{y = 1\}$ and `send(b)` with $\{z = 0\}\mathtt{send(b)}\{z = 1\}$. Finally, we define

[1] In the classical Owicki-Gries framework this is directly forbidden by the interplay of the syntactic rules of the usage of the global variable and the side conditions of the axioms for CR and parallel composition.

$G \equiv y := 1$ for send(a) and $G \equiv z := 1$ for send(b), to update y and z, respectively, at line 8 in Listing 6.1 when send is executed.

With verification axioms similar to Owicki-Gries, it can be proved that these assertions indeed confirm the correctness of the client property. In particular, the conjunction of the postconditions of send(a) and send(b), and I_A, i.e., $y = 1 \wedge z = 1 \wedge len(\mathtt{q}) = y + z$, implies the desired client postcondition $len(\mathtt{q}) = 2$.

Passing corresponding ghost codes G to instances of m allows for abstraction and parallelism, but it does not make the approach modular. Each context and/or property likely requires different ghost variables, and hence different P, Q, I_A, and G. Suppose that we want to verify a property about the content of q using a function *cnt* mapping the queue content to a set of messages. Specifically, we want to prove that if in the beginning, $cnt(\mathtt{q}) = \emptyset$, then at the end, $cnt(\mathtt{q}) = \{\mathtt{a}, \mathtt{b}\}$. In this case, our ghost variables range over sets of messages, and the specifications must be adjusted accordingly, i.e., $\{y = \emptyset\}\mathtt{send(a)}\{y = \{\mathtt{a}\}\}$, $\{z = \emptyset\}\mathtt{send(b)}\{z = \{\mathtt{b}\}\}$, $I_A \equiv cnt(\mathtt{q}) = y \cup z$, and $G \equiv y := \{\mathtt{a}\}$ and $G \equiv z := \{\mathtt{b}\}$ for send(a) and send(b), respectively. Even if we had a library of predicate sets and ghost code blocks, in general we would not be able to cover all possible contexts in which the generic code, i.e., m, could be used.

Greater generality can be achieved by a schema along the lines of [67] in which P, Q, I_A, and G are parameters of the specification of m. The schema imposes some constraints on these parameters which become proof obligations when verifying code involving m. Under these constraints, m needs to be verified only once. For each new context, the client only needs to verify that the constraints hold. We propose a new modular specification schema (MSS) that allows further abstraction from the implementation details of m, by supporting parameterization based on CRs. Unlike in [67], the semaphores that implement the CR as well as the names of the fields that determine the state of the object (s and q, resp., in the send example) remain absent from the specification. As a result one retains the flexibility of the OO approach. For example, if the implementation of the CR is changed such that locks are used instead of semaphores, the specification can remain the same.

We proceed by giving the intuition behind the MSS. We first establish the relationships between the parameters P, Q, I_A, and G, that need to hold in order for the specification to be correct. Later we lift these relationships to the level of the whole method m to formulate the MSS.

Listing 6.3: A semaphore based implementation of a CR

```
1  {P}
2  s.acquire()
3  {IA(s) * P}
4  {O(v) * I(v) * P}
5  C
6  {O(post(v)) * I(v) * P}
7  G
8  {O(post(v)) * I(post(v)) * Q}
9  {IA(s) * Q}
10 s.release()
11 {Q}
```

Assume that the body of m consists of only a single CR implemented by using semaphore `s`. The CR is of the form `s.acquire()` C `s.release()` as given in Listing 6.3. Without loss of generality, let us assume that the CR protects a single field `f` of an instance `o` of class `C`. Field `f` can be changed only within the CR.

When establishing the relationships, the separating conjunction operator $*$ introduced in Section 2.3 is used. We are guided by the correctness requirements for the annotation of Listing 6.3 in the familiar Hoare logic/Owicki-Gries style. The validity of P and Q at lines 1 and 11, respectively, implies that $I_A(\mathtt{s}) * P$ and $I_A(\mathtt{s}) * Q$ hold at lines 3 and 9, respectively (we write $I_A(\mathtt{s})$ instead of just I_A to emphasize that it is associated with `s`). This is analogous to the proof rule for the CR in Owicki-Gries, which follows from the rules below (for `acquire` and `release` methods of a semaphore combined with the frame rule)

$$\{\mathbf{emp}\}\ \mathtt{s.acquire()}\ \{I_A(s)\} \tag{SA}$$

$$\{I_A(s)\}\ \mathtt{s.release()}\ \{\mathbf{emp}\} \tag{SR}$$

and the fact that P and Q do not refer to `s` and hence only involve parts of the heap disjoint from the parts affected by `acquire` and `release`.

To capture the environment constraints, next to ghost variables, $I_A(\mathtt{s})$ may also depend on `o.f`. To avoid directly referring to `f`, we introduce a so-called *payload invariant* I, parameterized with a ghost variable v, thereby making the approach modular. In the example of Listing 6.2, $I_A(\mathtt{s}) \equiv \mathit{len}(\mathtt{q}) = y + z$ would be substituted by $I(v) \equiv \mathit{len}(v) = y + z$. To link the actual field `f` with its ghost counterpart v we use predicate $O(v)$ (for the earlier `send` example, we could define $O(v) \equiv \mathtt{q} = v$). $O(v)$ is an abstract predicate local to `o` that is not visible for the client. By defining $I_A(\mathtt{s}) = \exists v.O(v) * I(v)$, we circumvent the need to refer to `o.f` in the client invariant.

Line 4 in Listing 6.3 is obtained by substituting $O(v) * I(v)$ for $I_A(s)$ at line 3. Since C affects only actual variables, P holds also in the postcondition of C at line 6. However, since the actual variables have changed while ghost variable v remains the same, predicate O holds only for an adjusted value of v given by $\mathit{post}(v)$. In our example, $\mathit{post}(v) \equiv \mathit{len}(v) + 1$. G only affects y and z, so after G, $O(\mathit{post}(v))$ remains valid. So, in order to recover the invariant I_A, G at line 7 should be chosen such that it modifies the ghost variables occurring in $I(v)$ and P in such a way that $I(\mathit{post}(v))$ becomes **true** and P is transformed to Q (line 8). Proving that G indeed has this property remains a proof obligation for the client program calling m and as such becomes a premise of our schema. It is easy to check that this constraint is satisfied by all instances of `send` in the running example for both client properties. Finally, line 9 follows directly from line 8 by the definition of $I_A(\mathtt{s})$.

6.2.2 Formulating the MSS

By summarizing the constraints on the various elements of the annotation, and lifting them to the level of method m, we obtain the MSS:

$$\frac{\forall v \bullet \{P * I(v)\}\ G\ \{Q(\mathit{res}(v)) * I(\mathit{post}(v))\}}{\{[\pi]\mathtt{o}.A(I) * P\}\ \mathtt{r} := \mathtt{o}.m(G)\ \{[\pi]\mathtt{o}.A(I) * Q(\mathtt{r})\}}$$

For simplicity, we assume that m has no parameters besides G. However, additional parameters can be captured in the usual way for procedure verification rules in Hoare logic. We also assume that m returns a result $res(v)$ immediately after leaving the CR, that is assigned to variable `r`. In general, Q depends on `r`. Both $res(v)$ and $post(v)$ are fixed by the supplier of m. Also recall that we define the semaphore invariant $I_A(\texttt{s}) = \exists v.O(v) * I(v)$, which allows us to omit the occurrence of v in the schema under the line.

Predicate A links the semaphore used to implement the CR inside method m with the payload invariant I. A is an abstract predicate in the sense that the client does not need to know its definition since it is local to `o`. For the `send` example, A would state that there is a semaphore `s` that is properly initialized and it associates to A a semaphore invariant $I_A(\texttt{s})$ (formed using $I(v)$ as described earlier). These implementation details, including `s`, are hence not visible to the client calling m. Finally, π is an arbitrary fraction denoting a fractional permission for A.

Like the schema in [67], MSS is not an axiom or a proof rule of the proof system and it does not affect the soundness of the proof system, since for any correct module it can be derived from other axioms and rules. Specifically, each module can be specified separately based on MSS and its implementation can be annotated in a way following from the proof outline in Listing 6.3. As MSS is a derived proof rule, the implementation of each module still needs to be verified by a proof system against its specification. In our case, the proof system is separation logic.

MSS can be seen as a means to divide the proof obligations between the client and the supplier of m. The schema is implicitly universally quantified over P, Q, I, and G. Note that *post* and *res* are fixed by the supplier and that they implicitly define the effect of C on `o.f` in a sequential environment. On the other hand, the client is free to use any predicates P, Q, I, and G satisfying the premise of MSS. For any such predicates, the supplier guarantees that the implementation of m satisfies the triple in the consequent of MSS.

The premise of MSS $\forall v.\{P * I(v)\}\ G\ \{Q(res(v)) * I(post(v))\}$ is analogous to the premise of the Owicki-Gries CR axiom $\{P * I_A(\texttt{s})\}\ C\ \{Q(\texttt{r}) * I_A(\texttt{s})\}$. MSS, however, shifts the verification from the actual code C and invariant I_A to the ghost code G and the payload invariant I. Although C does not appear in MSS, its specification is reflected in v, $post(v)$ and $res(v)$. Although G has to reflect all important aspects of each call of `o.m`, the method is still to a great extent modular since the implementation and verification of the program text of $o.m$ remains completely independent of calling of `o.m`.

6.3 Implementing the SLCO Channel

In Section 3.4 we discussed two possible implementations of SLCO asynchronous channels. One implementation covers partial features of SLCO asynchronous channels while the other covers all their features. Specifically, the simplified implementation does not support blocking sending and receiving operations. Also, conditional receiving operations of SLCO asynchronous channels are not supported. The advantage of the simplification is that it facilitates the specification and verification of the implementation.

In this section we present the simplified implementation of SLCO asynchronous channels. As shown in Listing 6.4, the model-generic part of SLCO asynchronous channels is implemented as a Java class `Channel` which contains three fields.

Listing 6.4: The implementation of the Channel class

```
1 public final class Channel {
2   List itemList;
3   Semaphore s;
4   int queueMaxSize;
5   public Channel(int queueMaxSize)
6   {
7     itemList = new ArrayList();
8     this.queueMaxSize = queueMaxSize;
9     s = new Semaphore(1);
10  }
11  boolean send(String msg) {
12    ...
13  }
14  String receive() {
15    ...
16  }
17 }
```

As shown at line 2 in Listing 6.4, the list `itemList` is used to implement the buffer which is associated with SLCO asynchronous channels. Semaphore `s` at line 3 is used to implement access to the CR within the operations and `queueMaxSize` at line 4 defines the maximum channel capacity. The method `send` at lines 11-13 is used to wrap sending operations of SLCO asynchronous channels while the method `receive` at lines 14-16 is used to wrap receiving operations of SLCO asynchronous channels. The implementations of methods `send` and `receive` are shown in Listing 6.5 and Listing 6.6, respectively. The messages passed through channels are processed in FIFO order when the maximum channel capacity is greater than 1, i.e., messages are added to the end and removed from the front of `itemList`.

The implementation of the `send` method of class `Channel` is shown in Listing 6.5. The `send` operation has one parameter `msg` (in line 1), the message that is being sent. At line 2, semaphore `s` is acquired for mutual accesses between multiple threads. At lines 3-4, the channel is checked whether it is full. If it is not full and the current content of the channel is `q`, then after execution of `send` the content of the channel is `q + msg`, where + denotes concatenation of sequences of messages. If the channel is already full, the content is unchanged. Furthermore, `send` returns a Boolean result (at line 9) indicating whether or not the operation was successful. The `send` method either returns **true** on successful sending operations or returns **false** on unsuccessful sending operations to its caller. Correspondingly, the execution of the caller is not blocked at the execution of any code within the `send` operation, which is in line with the simplified implementation of SLCO asynchronous channels.

Listing 6.5: The implementation of the send method of class Channel

```
1 boolean send(String msg) {
2   s.acquire();
3   boolean result = itemList.size() < queueMaxSize;
4   if (result) {
```

```
5       itemList.add(msg);
6       result = true;
7     }
8     s.release();
9     return result;
10  }
```

The implementation of the `receive` method of class `Channel` is shown in Listing 6.6. At line 2 semaphore `s` is acquired before accessing list `itemList`, shared between multiple threads. At line 4, the channel is checked whether it is empty. If the channel is empty, then the value `null` is returned to its caller, as shown at lines 4-6. If the channel is not empty and the channel has contents `msg + q`, then message `msg` is removed from the list `itemList` at line 7. Specifically, the channel's new contents after execution of `receive` is `q`, and message `msg`, i.e., the first element from the front of the list `itemList`, is returned as a result. Similar to the `send` operation in Listing 6.5, the `receive` operation is non-blocking operation, as it returns either `null` or the first element from the front of the list `itemList`. As the simplified implementation does not consider the conditional receiving operations, the content of the message `msg` is not checked before removing it from `itemList`.

Listing 6.6: The implementation of the receive method of class Channel

```
1  String receive() {
2    s.acquire();
3    String result;
4    if (itemList.size() == 0) {
5      result = null;
6    } else {
7      result = (String)itemList.remove(0);
8    }
9    s.release();
10   return result;
11 }
```

6.4 Specifying and Verifying the SLCO Channel

In the previous section we demonstrated the simplfied implementation of SLCO asynchronous channels that can hold a predefined maximum number of messages. In this section we present its specification and verification for use in a generic, multi-threaded environment via separation logic in VeriFast. Using VeriFast, we verify the absence of race conditions. Besides this, we also show how to prove properties of clients using the channel. Furthermore, our modular approach described in Section 6.2 is applied to the specification and verification.

The VeriFast specification of the `send` method in Listing 6.5 following from MSS is given in Listing 6.7. Predicates `A`, `I`, `P`, and `Q` correspond to their namesakes in the MSS, whereas the assertion `is_G_S` implements the passing of the ghost code G into the method. Both `[?pi]` and `[pi]` correspond to the fractional permission $[\pi]$. The question

mark `?` in front of a variable means that the value of the variable is recorded and that all later occurrences of that variable in the contract have the same value which is equal to the value of the first occurrence. For instance, in Listing 6.7, the value of the fractional permission `pi` in the precondition must be the same as the one in the postcondition (as also required in the MSS).

Listing 6.7: Part of the channel specification

```
1 public final class Channel {
2   //...
3   boolean send(String msg)
4   /*@
5   requires [?pi]A(?I) &*& is_G_S(?G, this, I, msg, ?P, ?Q)
       &*& P();
6   ensures [pi]A(I) &*& Q(result);
7   @*/
8   // ...
9 }
```

Predicates `P`, `Q` and `is_G_S` are left undefined and are supposed to be provided by the client. More precisely, a *lemma function* `G` is supplied by the client based on which VeriFast automatically creates the predicate `is_G_S`.

Listing 6.8 contains the specification of a lemma function `G` that corresponds to the ghost statement block G. Lemma functions in VeriFast are methods without side effects which help the verification engine. The contract of a lemma function corresponds to a theorem, its body to the proof, and a lemma function call to an application of the theorem. Note that the specification of `G` in Listing 6.8 corresponds to the premise of MSS, where $post(v)$ specifies that if $res =$ **true**, `msg` has been added to the channel, and otherwise it has not (line 4).

Listing 6.8: A lemma function specifying the ghost statement block G

```
1 /*@
2 typedef lemma void G (Channel c, predicate(list<Object>, int
    ) I, String msg, predicate() P, predicate(boolean) Q) (
    boolean res);
3   requires P() &*& I(?items, ?qms);
4   ensures  Q(res) &*& I(res ? append(items, cons(msg, nil))
      : items, qms);
5 @*/
```

Method `send` is part of the class `Channel` (Listing 6.9), implementing the SLCO channel construct. Class `Channel` contains three fields: the list `itemList` implementing the FIFO queue, semaphore `s` that is used to implement access to the CR within the operations, and `queueMaxSize` defining the maximum channel capacity. For verification purposes we add the ghost field `inv` which is used to keep track of the invariant.

Semaphore invariant `I_A`, corresponding to I_A in Section 6.2, is given at line 2 in Listing 6.9. The invariant is defined by means of a predicate constructor parameterized with the payload invariant `I`. Corresponding to the definition of I_A, in `I_A`, it is checked that for ghost variables `items` and `qms`, i.e., the contents of the item list and the maximum

number of messages, respectively, `I` holds. The question mark `?` is used to record the value of the variable following it, for use later on in the predicate. Operator $\mapsto$ is written in VeriFast as `|->`, and the expression of the form `[f]` denotes fractional ownership with fraction `f`. When `f` = 1, the fractions are omitted, and an arbitrary fraction is denoted as `[_]`.

Listing 6.9: The specification of the Channel class

```
1 /*@
2 predicate_ctor I_A(Channel channel, predicate(list<Object>,
      int) I)() = channel.O(?items, ?qms) &*& I(items, qms);
3 @*/
4
5 public final class Channel {
6   List itemList;
7   Semaphore s;
8   int queueMaxSize;
9   //@ inv inv;
10  //@ predicate O(list<Object> items, int qms) = this.
      itemList |-> ?itemList &*& itemList.List(items) &*&
      this.queueMaxSize |-> qms &*& length(items) <= qms;
11  //@ predicate A(predicate(list<Object>, int) I) = ... &*&
      s |-> ?sem &*& [_]sem.Semaphore(I_A(this, I));
12 }
```

In predicate `O` (line 10), the links are established between ghost variables and fields. The first conjunct in this line `channel.itemList |-> ?itemList` implies exclusive ownership of the field `itemList` and at the same time that the value of `itemList` is recorded for later use in the contract. Expression `itemList.List(items)` states the fact that `itemList` is a list with elements `items`. The final conjunct links `queueMaxSize` to ghost variable `qms`.

We use the VeriFast ownership concept to implement syntactic restrictions in the Owicki-Gries approach on the variables. In particular, we need to ensure that the fields like `itemList` can be modified only in the CR implemented by semaphore `s` and that the ghost variables are modified exclusively by at most one method, in this case `send`.

Predicate `A` is given at line 11 in Listing 6.9. Like its MSS counterpart A, it is parameterized with the payload invariant `I` (corresponding to I in MSS). Besides some auxiliary conjuncts, it has two conjuncts to associate `I_A` with `s`. The second conjunct is parameterized with the object itself and the payload invariant.

Listing 6.10 contains the `send` method with its corresponding full annotation that further facilitates verification. Since VeriFast does not automatically unfold predicate definitions, ghost statement `open` is used to do this, i.e., to replace the predicate with its definition. In this way the heap chunks of the definition are made visible to the verifier. The opposite effect is achieved by `close` which replaces heap chunks with the corresponding predicate definition. At line 6 predicate `A` is unfolded to obtain the predicates needed for acquiring `s`. After the acquisition of the semaphore also its invariant `I_A` is opened at line 9 to get access to the heap chunks related to `itemList` and `queueMaxSize`.

The code segment at lines 11-13 corresponds to C in the MSS, and affects the "real" variables. The code at lines 15-17 is ghost code. The lemma function performing the

updates of the ghost variables is called at line 15. Annotation of the `receive` method can be done in an analogous way.

Listing 6.10: The annotation of the Channel `send` method

```
1 public final class Channel {
2   //...
3   public boolean send(String msg)
4   /*@ requires ... ensures ... @*/
5   {
6     //@ open [pi]A(I);
7     //@ s.makeHandle();
8     s.acquire();
9     //@ open I_A(this, I)();
10
11    boolean result = itemList.size() < queueMaxSize;
12    if (result)
13      itemList.add(msg);
14
15    //@ G(result);
16    //@ length_append(items, cons(msg, nil));
17    //@ close I_A(this, I)();
18    s.release();
19    //@ close [pi]A(I);
20    return result;
21  }
22  //...
23 }
```

Class `Channel` annotated as in Listing 6.10 is verifiable against its specification in VeriFast. This means that it is free of deadlocks and race conditions. Those requirements are not explicitly specified, but are always checked when VeriFast tries to verify code. The class is now ready to be used by client programs to verify specific properties, using the pre- and postconditions and the payload invariant.

Next, we discuss how the property 'if k messages are sent over the channel, k messages will be received' can be specified for a program using the channel via one sending and one receiving thread. First of all, Listing 6.11 specifies the client program we use. In the `main` method (lines 6 and onwards), an instance of the channel is created, and a sending and a receiving thread are started, one sending k, i.e. `messageMaxCount`, messages, and the other one trying to receive them. To specify the property, we introduce two new ghost variables for counting the number of messages (lines 2 and 3). In the precondition of `main`, we require that the class has been properly initialized (conjunct 1 at line 7), link the `messageMaxCount` variable to the ghost variable `mmc`, and have an additional requirement that it is at least equal to 1. In the postcondition, we link `sendCount` and `receiveCount` respectively to `sc` and `rc`, and require that they are both equal to `mmc` (line 8).

Listing 6.11: Client program specification

```
1 public class Program {
```

```
2   //@ static int sendCount;
3   //@ static int receiveCount;
4   public static int messageMaxCount; // k
5
6   public static void main(String[] args)
7   //@ requires class_init_token(Program.class) &*&
       Program_messageMaxCount(?mmc) &*& 0 < mmc;
8   //@ ensures Program_messageMaxCount(mmc) &*& [_]
       Program_sendCount(?sc) &*& [_]Program_receiveCount(?rc)
        &*& mmc == sc &*& mmc == rc;
9   {
10    // ...
11  }
12 }
```

A crucial role in the verification using MSS is played by the client (payload) invariant I shown in Listing 6.12. The invariant states that at any point of each program execution the number of received messages cannot exceed the number of sent messages. The signature of `I` (line 4) complies with the specification of the corresponding abstract predicate in Listings 6.8 and 6.9 and it has two arguments. The first argument is a list containing the messages that are in the channel queue. The second argument, corresponding to the maximal number of messages in the queue, is actually not used in the definition of this concrete instance of I. At line 5 the first conjunct states that all messages in the queue are not null. Lines 6 and 7 are used to link the mathematical variables `sc` and `rc` with the ghost variables `sendCount` and `receiveCount` that we added to class `Program` in Listing 6.11. Similarly, at line 8 program variable `messageMaxCount` (Listing 6.11) is linked to the mathematical variable `mmc`. In Lines 9-11 various relations between the mathematical variables are expressed. Finally, at line 12 the main claim of the invariant is given, i.e., that the number of received messages is not greater than the number of sent messages.

Listing 6.12: Client (payload) invariant I

```
1 /*@
2 fixpoint boolean non_null(Object o) { return o != null; }
3
4 predicate I(list<Object> items, int qms) =
5   forall(items, non_null) == true &*&
6   [1/2]Program_sendCount(?sc) &*&
7   [1/2]Program_receiveCount(?rc) &*&
8   [1/3]Program_messageMaxCount(?mmc) &*&
9   0 <= sc &*& sc <= mmc &*&
10  0 <= rc &*& rc <= mmc &*&
11  length(items) == sc - rc &*&
12  rc <= sc ;
13 @*/
```

To determine that the postcondition holds, we need to specify the thread sending the messages. In Listing 6.13, at lines 2 and 3, its pre- and postcondition are specified. In the

run method, the messages are sent. For the send call at line 27, we need to provide ghost code G_S. This is done in lemma ghost_send, where the ghost variables are updated. This lemma is linked to the call at line 25. The pre- and postcondition of send are specified as two predicates, P and Q, see lines 16 and 17.

Listing 6.13: SenderThread class specification

```
1  class SenderThread implements Runnable {
2    //@ predicate pre() = this.c |-> ?c &*& [_]c.A(I) &*& [_]
       Program_sendCount(0) &*& [_]Program_messageMaxCount(?
       mmc) &*& 0 < mmc;
3    //@ predicate post() = this.c |-> ?c &*& [_]c.A(I) &*& [_]
       Program_messageMaxCount(?mmc) &*& [_]Program_sendCount
       (?sc) &*& mmc == sc;
4
5    Channel c;
6    ...
7
8    public void run()
9    //@ requires pre();
10   //@ ensures post();
11   {
12     for (i = 0; i < Program.messageMaxCount; i++)
13     {
14       for (;;){
15       /*@
16         predicate P() = [1/2]Program_sendCount(i) &*& [1/3]
             Program_messageMaxCount(mmc);
17         predicate Q(boolean r) = [1/2]Program_sendCount(r ?
             i + 1 : i) &*& [1/3]Program_messageMaxCount(mmc);
18         lemma void ghost_send(boolean r)
19         requires ... ensures ...
20         {
21           open P();
22           ...
23         }
24       @*/
25       //@ produce_lemma_function_pointer_chunk(ghost_send) :
             G_S(c, I, m, P, Q)(r) { call(); };
26       //@ close P();
27       boolean success = this.c.send("message");
28       //@ open Q(success);
29       }
30     }
31     //@ close post();
32   }
33 }
```

VeriFast was able to verify the code against its specification, meaning that the property

holds. Besides the environment with two threads, an environment consisting of multiple senders and receivers was verified in [87] following from the schema in [67]. We adapted also this work to MSS, but did not include it in this thesis. The definitions of the various predicates, including the payload invariant, are similar to the ones given above for the case of one sender and one receiver.

6.5 Related Work

An approach to generate Java code from Communicating Sequential Processes (CSP) specifications is described in [106]. The authors describe how they have verified that a CSP model of their implementation of a channel semantically corresponds to a simpler CSP model describing the desired functionality of that channel. First of all, by working from a model describing the implementation, as opposed to the implementation itself, one still needs to prove that the model corresponds exactly to the implementation to establish that the implementation itself is correct. Moreover, it seems that a fully modular verification approach in the way we wish to have it is not completely possible; for instance, although it would be possible to use their simpler CSP model of a channel within detailed implementation-level CSP models of systems using channels, one could not abstract away the functionality of a channel to the same extent as when using separation logic if one would like to prove a functional property referring to communication, but not expressing how the communication itself should proceed.

Regarding theorem proving, to the best of our knowledge the approach in [67] was the first one supporting fully general modular specification and verification of fine-grained concurrent modules and their clients. Compared to the schema in [67], the MSS we propose imposes conditions on the ghost code instead of the actual code, and abstracts away the implementation of the protected object better than [67] does, thereby improving the modular nature of the approach.

An approach comparable to [67] appears in [97] where a new separation logic is presented with concurrent abstract predicates. Furthermore, in [96] the authors have applied their approach to prove correctness of some synchronisation primitives of the Joins concurrent C# library. As far as we know, the authors do not intend to eventually use their approach to verify model transformations. It remains to be investigated whether theirs can be used for that as well.

Adding ownership types [49,118] to Java is a very effective technique to verify that Java threads always access data correctly, i.e. for which they have acquired the proper access rights. Such a technique offers an alternative way to verify that our channel implementation is always correctly accessed. However, it cannot be used to verify arbitrary functional properties that may rely on ownership, but express more than that, such as that some desired behaviour is guaranteed to always eventually happen. On the other hand, with separation logic, one can express and verify such properties as well.

6.6 Conclusions and Future Work

In this chapter, we presented a simple Java implementation of SLCO asynchronous channels as a generic part of the model-to-code transformation from SLCO to Java. This implementation does not cover the condition reception feature, as discussed in Section 3.4. Also,

the implementation itself does not support blocking sending and receiving operations that wait for the corresponding buffer to become non-empty when receiving a message, and wait for space to become available in the buffer when sending a message. The advantage of this simplification is that it facilitates the verification of the implementation. A more complete implementation of SLCO asynchronous channel as well as SLCO synchronous channel was presented in Chapter 4, which incorporates more advanced features of Java. For future research, the specification and verification of the implementation of SLCO asynchronous channels presented in this chapter can be extended to support the more complete implementation.

We also provided the verification of the simple implementation of SLCO asynchronous channels via a modular approach for the verification of fine grained concurrent code. The ideas behind and the feasibility of such an approach is demonstrated as well. With its support of parameterized verification, concurrency via threads, object-oriented code, and fast verification results, VeriFast was up to the task - though an experienced user is required. This underlines the relevance of the idea of re-using generic code that has to be verified only once.

Additionally, we proposed a novel modular specification schema which improves the modularity of the VeriFast approach. Although our schema was developed using separation logic and VeriFast, it can be straightforwardly adapted for the standard Owicki-Gries method (assuming extensions with modules) or similar formalisms for concurrent verification.

Besides verifying the correctness of SLCO channel implementations, an important robustness criterion of dependency safety regarding sending and receiving operations of a channel also needs to be considered in the implementation. That is, when a thread fails when executing the sending operation of a channel, it is desired that its corresponding receiving operation operated by another thread is not executed anymore nor will wait for the completion of the sending operation. To this end, we will introduce a language extension called *failbox* in the next chapter. This language extension helps to ensure that a program satisfies the important robustness criterion of dependency safety.

Chapter 7

Increasing Robustness via Failboxes

During execution of a multi-threaded program generated from an SLCO *model, when a thread fails when executing a send operation of a channel, it is desired that the corresponding receive operation operated by another thread is not executed anymore, and that receive operation currently executing no longer waits for the completion of the send operation. This can be achieved by using a language extension called failbox. It helps to ensure that a program satisfies the important robustness criterion of dependency safety: if an operation fails, no code that depends on the operation's successful completion is executed anymore or will wait for the completion of that operation. However, the original implementation of failbox is in Scala, whereas for our setting, we require a Java implementation. Moreover, it requires the assumption of absence of asynchronous exceptions inside the failbox code. To address these issues, in this chapter we provide a Java implementation without this assumption. The assumption is eliminated in an incremental manner through several increasingly more robust implementations which are all presented in this chapter.*

7.1 Introduction

In Chapter 6, we indicated that during execution of a multi-threaded program generated from an SLCO model some operations, such as send and receive operations of a channel, may fail. A way to mitigate the effect of such failures is to ensure that the program has the property of *dependency safety* [66]: if an operation fails, no code that depends on the operation's successful completion is executed anymore or will wait for the completion of that operation.

As argued in detail in [66], the exception handling mechanisms available in languages like Java [51] do not provide a direct way to achieve, let alone verify, dependency safety. Problems on the safety side [66] are that exiting a `try` block (as part of a `try-catch` statement) may leave a data source in a corrupted state. Problems on the liveness side [63] are that a `wait` statement may wait on the successful return of a failing operation, causing that thread to wait indefinitely. An example of the latter, taken from [63], is the Scala [4] program in Listing 7.1. There, the main thread indefinitely blocks on `take` (line

3) if the forked thread, launched at line 2, fails.

Listing 7.1: Motivating example

```
1 val queue = new LinkedBlockingQueue[String]()
2 fork { queue.put("Hello, world") }
3 queue.take()
```

A concept to achieve dependency safety in Java, called *failbox*, is introduced in [66]. This language extension allows dealing with exceptions compositionally. The mechanism of failboxes is as follows. All threads that depend on each other's successful execution of instructions are forced to run in the same failbox. As soon as an instruction executing in a failbox fails (leading, e.g., to its thread terminating abruptly with an unchecked exception), all threads in the same failbox will be notified, and terminated.

To achieve dependency safety, if an operation B depends on an operation A it must be ensured that they execute in the same failbox.

An example how one can use the failbox mechanism to fix the problem with the program in Listing 7.1 is given in Listing 7.2.

Listing 7.2: Proposed fix with Failbox

```
1 fb = new Failbox()
2 fb.enter {
3   val queue = new LinkedBlockingQueue[String]()
4   fork { fb.enter{queue.put("Hello, world")} }
5   queue.take()
6 }
```

Since the main and the forked thread run in the same failbox, if the forked thread crashes the failbox mechanism ensures that the main thread is notified about it. As a result, unlike in Listing 7.1, the main thread no longer blocks on `take`. Instead, it will be informed about the failure and terminate.

There are several technical limiting assumptions with respect to the actual failbox implementations provided in [63, 66] which are related to the occurrence of *asynchronous exceptions*. Unlike synchronous exceptions (such as null pointer and array out of bounds) that occur when executing a specific statement, asynchronous exceptions can happen anywhere in the program. Hence, asynchronous exceptions cannot be completely dealt with by `try-catch` statements. These exceptions are caused by external factors, like the user interrupting (a part of) the program or another thread sending a stop signal.

The issues with asynchronous exceptions need to be resolved to guarantee dependency safety without limiting assumptions. This chapter presents solutions to these problems in an incremental manner, and provides better insight into the subtle issues involved.

Our main contributions are as follows:

1. In [63] a failbox implementation in Scala was provided, which is used to translate to Java in this chapter. This implementation is very intuitive because Scala turned out to be particularly amenable to implementing the failbox concept. As stated in [63], this implementation provides the failbox functionality under the assumption that *no asynchronous exceptions occur in certain parts of the code implementing the failbox.*

   To better connect to Java and Java-tooling, a Java implementation is desirable. This is not a straightforward matter, because Java differs in various aspects from Scala

that make the latter so suitable for implementing the failbox. A Java implementation that matches the Scala implementation, with the same limiting assumptions, is presented in Section 7.2.

2. It turns out that using Java's mechanism of uncaught exception handlers, the assumptions can be weakened to only concern the *handlers of the failbox code*. An analysis concerning this observation and an implementation based on it is presented in Section 7.3. The analysis leads to the conclusion that, to further lift the limiting assumption, it might be necessary to use the Java Native Interface (JNI) [3].

3. An implementation pushing the uncaught exception handler approach to its limits, using JNI, that almost completely removes the assumption is provided in Section 7.4.

4. Finally, a precise analysis of the remaining restriction leads to a JNI level solution that does not use the uncaught exception handler and which is fully hardened against asynchronous interrupts; this is provided in Section 7.5.

In Section 7.6 we discuss the related work, and in Section 7.7 we draw conclusions and give directions for further research. In this chapter, we focus on implementations of failboxes. In Chapter 8, we present a testing approach for demonstrating the dependency safety weaknesses of the different failbox implementations in this chapter. Besides that, a further interesting step which is not included in this thesis is to specify and verify the last and most robust implementation. Currently, we have been working on specifying and verifying its simplified version. The implementation is simplified in the sense that the code for notifying other threads has not been considered yet. This simplification facilitates the verification of the failbox's implementation. The next step is to specify and verify the full implementation.

7.2 A Basic Failbox Implementation

In this section, we describe the first iteration of our failbox implementation. It is based on the Scala implementation from [63]. Like its Scala counterpart, it is still vulnerable to asynchronous interrupts. It can be seen as a stepping stone towards the more robust implementations presented in subsequent sections.

A failbox is basically an object with a boolean variable, telling whether the failbox is operational (i.e., has failed or not), and a list of threads. Furthermore, it provides the method `enter`, which is called by a thread to add itself to the failbox, passing as an argument the code to be executed. If this code fails, the `enter` method notifies all other threads in the failbox. The Java implementation is shown in Listing 7.3.

Listing 7.3: Java implementation of a Failbox

```
1 class FailboxException extends RuntimeException {}
2 class Failbox {
3   private boolean failed;
4   private ArrayList<Thread> threads;
5   public void enter(Thread currentThread, Runnable body){
6     synchronized (this) {
7       if (failed) throw new FailboxException();
```

```
8         threads.add(currentThread);
9       }
10      try {
11        try {
12          body.run();
13        } finally {
14          synchronized (this) {
15            threads.remove(currentThread);
16          }
17        }
18      } catch (Throwable t) {
19        synchronized (this) {
20          failed = true;
21          for (Thread tr : threads) {
22            tr.interrupt();
23          }
24        }
25        throw t;
26      }
27    }
28  }
```

In Listing 7.3, the boolean variable `failed` (line 3) has initial value `false` indicating that the failbox is operational. The ArrayList `threads` (line 4) contains all threads that are running within the same failbox.

Whenever a thread t needs to execute a code block and other threads depend on this succeeding, t calls the method `enter` of the failbox containing the other threads, passing the code block as a parameter wrapped in a Runnable object `body` (line 5). The method (lines 5-27) first checks the state of `Failbox` (line 7). If the `Failbox` has already failed for some reason, a `FailboxException` is immediately thrown. Otherwise, the current thread is added to the list `threads` (line 8). When t has finished executing the code block, either in a regular way or due to an error, it is removed from the list `threads` (lines 14-16). If an exception occurs while executing the code block, all threads in `threads` are notified (lines 19-24). Consistency of field `threads` is ensured via the `synchronized` mechanism.

Limitations analysis This basic implementation is not fully resistant to asynchronous exceptions. It may fail to notify its threads about crashes in two cases:

1. If an asynchronous exception happens when in `enter` before the outer `try` block is entered (lines 6-9) the `catch` part, which would notify the other threads, will not be executed.

2. If an asynchronous exception occurs in the `catch` part the notification may be interrupted.

Hence the limiting assumption: no asynchronous exceptions occur when execution in a failbox's `enter` method is before the outer `try` block or in the `catch` block.

7.3 An Implementation using Uncaught Exception Handler

The above mentioned vulnerabilities can be alleviated by using the mechanism of Uncaught Exception Handlers that was introduced in Java 1.5. Before terminating because of an uncaught exception the method `getUncaughtExceptionHandler` is called on the thread and the `uncaughtException` method is invoked with the thread and the exception as arguments.

The new `Failbox` class definition given in Listing 7.4 is an implementation of the `Thread.UncaughtExceptionHandler` interface.

Listing 7.4: Failbox class definition and construction

```
1 public class Failbox implements Thread.
    UncaughtExceptionHandler {
2   private boolean failed;
3   private ArrayList<Thread> threads;
4   Failbox() {
5     this.threads = new ArrayList<Thread>();
6     this.failed = false;
7   }
8 }
```

The fields `threads` and `failed` have the same meaning as before.

The `enter` method remains the core of the failbox implementation. In our context an important feature of the uncaught exception handler is that the method is executed no matter what happens with the thread that has set the handler. This is the last method that is being called before the thread is deleted. We use this feature to provide additional robustness to the failbox implementation. The code of the `enter` method is given in Listing 7.5.

Listing 7.5: The Failbox.enter method

```
1 public void enter(Thread currentThread, Runnable code){
2   currentThread.setUncaughtExceptionHandler(this);
3   synchronized (this) {
4     if (failed) throw new FailboxException();
5     threads.add(currentThread);
6   }
7   code.run();
8   currentThread.setUncaughtExceptionHandler(null);
9 }
```

The association between the thread and the failbox, the latter becoming its uncaught exception handler, is made in line 2 in Listing 7.5. We assume that this is the only way for each thread to set its handler, i.e., no other method except `enter` sets it. The part of the previous implementation of `enter` that notifies all other threads in the failbox is moved to the exception handler which we discuss shortly. This means that the processing of an interrupt is no longer confined to `try-catch` statements. Instead, the interrupts can be caught and processed as soon as they happen. Besides improving robustness, this makes

the implementation more efficient than the implementation in Listing 7.3 (and than its Scala counterpart in [63]). Note that `enter` should never be called within the scope of an exception handler.

The definition in Listing 7.4 requires an implementation of an uncaught exception handler method. Therefore, we provide one implementation of an uncaught exception handler method, as shown in Listing 7.6.

Listing 7.6: An Uncaught Exception Handler method

```
1 public void uncaughtException(Thread th, Throwable t) {
2   fail();
3 }
```

This simple method in Listing 7.6 forwards all occurrences of uncaught exceptions to the fail method provided in Listing 7.7.

Listing 7.7: The Failbox.fail method

```
1 public synchronized void fail() {
2   for (Thread tr : threads) { tr.interrupt(); }
3   threads.clear();
4   failed = true;
5 }
```

Similarly to the previous Java implementation, the Java version needs to interrupt all the threads inside the failbox.

Limitations analysis For the handler-based implementation, there are two cases where asynchronous exceptions can still cause problems.

1. The robustness of the failbox is improved, since crashes occurring while executing the statements at lines 6-9 in Listing 7.3 are now handled correctly. However, if an asynchronous exception happens in `enter` before the exception handler is set (line 2 in Listing 7.5) we have again the problem that the exception will be missed.

2. If an asynchronous exception occurs in the handler itself, the problem with the incomplete notification remains.

According to the above observations, the following assumption is still required: asynchronous exceptions do not occur when a thread is executing the code in the failbox `enter` method but has not yet set the exception handler, or when the handler is being executed.

7.4 An Implementation using Uncaught Exception Handler and JNI

The limitations mentioned at the end of the previous section can be partially removed by using native code (C or C++). Java supports native code via the Java Native Interface (JNI). Using the JNI framework, Java methods of an application running in a Java Virtual Machine (JVM) can call or be called by JNI functions.

By using the JNI, we can further improve the failbox. This is because the execution of JNI instructions is completely independent of the JVM execution. This means that the JVM cannot stop JNI methods in the middle of execution. By reimplementing the `enter` method in JNI, we obtain the guarantee that this method is executed regardless of the occurrence of uncaught interrupts. The code of the new `enter` method given in Listing 7.8 is a direct translation of the Java method in Listing 7.5, i.e., the corresponding Java calls are mapped to JNI calls.

The wrapper method calls with self-explanatory names in Listing 7.8 are used to retrieve the various class types, as well as the methods, fields, and field values of Java classes which were defined at the Java level. The retrieved elements are stored in variables of corresponding JNI types, like `jboolean` at line 5. For instance, the method `do_SetUncaughtExceptionHandler_method` at line 3 is used to wrap the call of Java method `setUncaughtExceptionHandler`. At line 5 the method `do_get_FailboxField_failed` is used to retrieve the value of the field `failed` of `failbox` and its return value is assigned to the variable `failed_value` which is defined as a `jboolean` type in JNI.

Listing 7.8: The Java_Failbox_enter C method

```
1 JNIEXPORT void JNICALL Java_Failbox_enter(JNIEnv* env,
    jobject failbox, jobject currentThread, jobject body)
2 {
3   do_SetUncaughtExceptionHandler_method(env, currentThread,
      failbox);
4   do_MonitorEnter(env, failbox);
5   jboolean failed_value = do_get_FailboxField_failed(env,
      failbox);
6   if (failed_value == JNI_TRUE) {
7     do_Throw_FailboxException(env, "FailboxFailed");
8   }
9   jobject threads = do_get_FailboxField_threads(env, failbox
      );
10  do_ArrayList_add(env, threads, currentThread);
11  do_MonitorExit(env, failbox);
12  if ((*env)->ExceptionCheck(env) == JNI_TRUE)
13    do_Failbox_fail(env, failbox);
14  do_Runnable_run(env, body);
15  if ((*env)->ExceptionCheck(env) == JNI_FALSE)
16    do_SetUncaughtExceptionHandler_method(env, currentThread
        , NULL);
17 }
```

The Java `synchronized` mechanism (line 6 in Listing 7.3) is replaced with JNI monitors to implement critical sections. A monitor is entered and exited by calling functions `do_MonitorEnter` (line 4 in Listing 7.8) and `do_MonitorExit` (line 11 in Listing 7.8), respectively. Within the monitor, the status of `failed` is checked and a possible exception is thrown. Furthermore, the current thread is added to the failbox. These operations are done in lines 4-11, corresponding to lines 3-6 in Listing 7.5.

Crucial for this implementation is the `ExceptionCheck()` method in line 12 in

Listing 7.8. It is responsible for checking in the JVM whether any exceptions have occurred during the execution. In this way, we can check if an asynchronous exception occurred before setting the exception handler.

The aforementioned functions `do_MonitorEnter` and `do_MonitorExit` are wrappers for the corresponding JNI functions `MonitorEnter` and `MonitorExit`. In the remainder of this chapter, we use the convention that a method named `do_`*name* is a wrapper for the corresponding JNI method *name*. All wrapper methods follow the pattern of the `do_MonitorEnter` method given in Listing 7.9.

If the call to the original JNI function *name* is unsuccessful, the native code will cause a crash. The `abort` method is called to terminate the whole program when the original JNI method *name* fails. For instance, the method `abort` is called in line 3 in Listing 7.9 when the call of JNI method `MonitorEnter` fails. If the call to the method `MonitorEnter` is successful, its return value is 0. Otherwise, its return value is a negative value.

Listing 7.9: The do_MonitorEnter method

```
1 void do_MonitorEnter(JNIEnv *env, jobject failbox){
2   jint result = MonitorEnter(env, failbox);
3   if (result != 0) abort();
4 }
```

In an analogous way, the code of the exception handler is implemented in JNI, see Listing 7.10, thereby further improving robustness. Compared to the Java version in Listing 7.7, the JNI implementation replaces the `synchronized` qualifier with the JNI `do_Monitor` to protect the shared fields. Also, after sending a notification to each thread in the failbox at line 8, its uncaught exception handler is unset at line 9.

Listing 7.10: The Java_Failbox_fail C method

```
1 JNIEXPORT void JNICALL Java_Failbox_fail(JNIEnv* env,
    jobject failbox)
2 {
3   do_MonitorEnter(env, failbox);
4   jobject threads = do_get_FailboxField_threads(env, failbox
      );
5   int arrayListSize = do_ArrayList_size(env, threads);
6   for (int i = 0; i < arrayListSize; i++) {
7     jobject arrayElement = do_ArrayList_get(env, threads, i)
        ;
8     do_Thread_interrupt(env, arrayElement);
9     do_SetUncaughtExceptionHandler_method(env, arrayElement,
        NULL);
10  }
11  do_ArrayList_clear(env, threads);
12  do_set_FailboxField_failed(env, failbox, JNI_TRUE);
13  do_MonitorExit(env, failbox);
14 }
```

Limitations analysis

1. The above discussion shows how Case 1 from the limitations analysis of the previous section is remedied.

2. Case 2 of the previous section is also covered to a significant extent. However, the JNI level approach presented in this section still retains one weak spot. To call the uncaught exception handler, one needs to briefly go back to Java, i.e., back to the JVM. This is because the uncaught exception handler, which calls the native code of `fail`, has to be written in Java, see Listing 7.6. If an asynchronous exception occurs in this method before the native implementation of `fail` has been started (line 2 in Listing 7.6), the remaining threads will not be properly notified.

Hence the limiting assumption: no asynchronous exceptions occur when executing the (Java) uncaught exception handler, before the native method `fail` has been started.

7.5 A JNI Implementation without Uncaught Exception Handler

To resolve the last remaining vulnerability of the JNI implementation with the uncaught exception handler, proposed in the previous section, we reconsider the incomplete initial solution in Java given in Listing 7.3. We translate the `enter` method of this Java implementation into JNI as given in Listing 7.11.

Listing 7.11: The enter JNI method of Class Failbox

```
1 JNIEXPORT void JNICALL Java_Failbox_enter(JNIEnv *env,
    jobject failbox, jobject currentThread, jobject body) {
2   do_MonitorEnter(env, failbox);
3   jboolean failed_value = do_get_FailboxField_failed(env,
      failbox);
4   if (failed_value == JNI_TRUE) {
5     do_Throw_FailboxException(env, "FailboxFailed");
6   }
7   jobject threads = do_get_FailboxField_threads(env, failbox
      );
8   do_ArrayList_add(env, threads, currentThread);
9   do_MonitorExit(env, failbox);
10  do_Runnable_run(env, body);
11  jthrowable exception = (*env)->ExceptionOccurred(env);
12  (*env)->ExceptionClear(env);
13  do_MonitorEnter(env, failbox);
14  do_ArrayList_remove(env, threads, currentThread);
15  if (exception != NULL) {
16    do_set_FailboxField_failed(env, failbox, JNI_TRUE);
17    int arrayListSize = do_ArrayList_size(env, threads);
18    for (int i = 0; i < arrayListSize; i++) {
19      jobject arrayElement = do_ArrayList_get(env, threads,
          i);
20      do_Thread_interrupt(env, arrayElement);
```

```
21      }
22    }
23    do_MonitorExit(env, failbox);
24    if (exception != NULL)
25    do_Throw(env, exception);
26  }
```

This JNI implementation closely follows its Java counterpart from Listing 7.3.

The `try-catch` statement is replaced by an exception check by means of the JNI method `ExceptionOccurred` (line 11) and a conditional statement (line 15). In particular, the inner `try` part (line 13) in Listing 7.3 is translated using the `do_Runnable_run` wrapper method (line 10). After checking exceptions, the exception flag is cleared (line 12) such that new exceptions can be registered. The `finally` part (lines 14-18) in Listing 7.3 is merged with the `catch` part of the `try-catch` statement into a critical section implemented using the wrapper methods already discussed in Section 7.4. In the critical section the current thread is removed unconditionally from the failbox (line 14). After that, if an exception has occurred (line 15), the original `catch` part is performed to set the `failed` flag (line 16) and the interrupts are sent to each of the threads of the current failbox (lines 18-21).

Since the native code cannot be interrupted by JVM (asynchronous) interrupts, this code ensures the correct termination of all threads in the failbox. Of course, a logical question that arises is "What if the execution of the native C code fails?". For example, if the execution dereferences a null pointer or if it overflows the C stack. This can cause problems if such a native exception is caught by the JVM and turned into a Java exception. However, the Java documentation does not mention such a transformation of exceptions. We reason that assuming the absence of "machine" exceptions in the C code is more acceptable than assuming that no exceptions occur while executing Java instructions. In any case, it seems inevitable that one must assume reliable execution at some level of the application.

7.6 Related Work

The failbox mechanism offers a solution to the dependency safety problem which is less involving in terms of programming effort, and induces less run-time overhead compared to manually guarding dependent code [65] and the technique based on the use of separate threads [13]. Our approach requires less effort than the alternatives requiring to always maintain consistency [46, 90] or never fail during critical sections [99]. For more elaborate arguments on this subject we refer the reader to [66]. Recent alternative approaches are the following. [17] specifies the flow of exceptions between modules with implicit invocation relations. [85] presents an error recovery technique that enables to apply runtime assertion checkers. [45] focuses on recovering to or achieving a consistent state using the transactional memory technique from [90]. [74] deals with single threaded programs strong exception safety, requiring recovery to the state before the unsuccessful operation started; extension to multi-threading is identified as challenging. [77] rather than grouping dependent threads as in our approach, uses intervals and happens-before relations to reason about which work belongs to the current phase of computation and which work belongs to the next phase of computation.

7.7 Conclusions and Future Work

We presented a Java implementation of the concept of failbox from [63]. Failboxes allow a simple way to deal with dependency safety problems in multithreaded programs. We presented four versions of the failbox implementation - two in Java and two in a combination of Java and C, using JNI. The assumption required in the original failbox implementation, i.e., absence of asynchronous exceptions inside the failbox code, is eliminated in an increasing manner through these four increasingly more robust implementations. For each implementation we analyzed the vulnerabilities and argue the remedies in the next implementation.

Translating the failbox to JNI and C also paves the way for an automated verification of the last and most robust implementation. In particular, this can be done using VeriFast.VeriFast does not currently support exception statements like `try-catch` in Java. However, they are not present in the C translation, which makes it possible to verify the C translation against its specification via VeriFast. The specification can also be used to prove dependency safety of code that uses the failbox. Also, the new capabilities of VeriFast [64] allow to verify absence of deadlocks, which is an important aspect of dependency safety [63].

The work presented in this chapter does not show the application of the failbox concept in our framework that transforms SLCO models to Java code. To be able to apply the failbox mechanism in the implementation, one needs to give a definition of all kinds of dependent operations including send and receive operations of channels at the SLCO model level. Based on the definition, an approach to analyze dependent operations of an SLCO model needs to be introduced when transforming the SLCO model to Java code. Correspondingly, dependent operations will be executed in the same failbox during the execution of the implementation.

Besides applying the failbox concept in the implementation, future work also involves formally verifying the proposed failbox implementations, and comparing the failbox approach to other mechanisms guaranteeing dependency safety. By setting objective criteria, a full evaluation of usability and performance could be performed.

In this chapter, for each implementation we analyzed the vulnerabilities and argued the remedies in the next implementation. A question that naturally arises for these incremental implementations is whether the vulnerabilities are realistic and also whether the remedies proposed are effective. In Chapter 7, we present a testing approach for demonstrating the dependency safety weaknesses of the different failbox implementations, thereby substantiating the claims in the current chapter.

Chapter 8

Test-Driven Evolution of Failboxes

In Chapter 7, we presented several possible implementations of the failbox concept, and claimed that they differ in the guarantees they provide regarding dependency safety. In this chapter, we present a testing approach for concurrent programs that enables to generate asynchronous exceptions in a controlled manner. The approach enables us to develop tests that demonstrate the dependency safety weaknesses of the different failbox implementations, thereby substantiating the earlier claims in Chapter 7. Furthermore, the tests are repeatable in that they give the same results for runs that may differ in scheduling, even on different platforms.

8.1 Introduction

An important requirement for concurrent programs is *dependency safety* [66]: if an operation fails, no code that depends on the operation's successful completion is executed anymore or will wait for the completion of that operation.

The failbox [66] is a language mechanism to achieve dependency safety for sets of threads in a modular manner. If a thread in a failbox fails, the failure will be detected and all other threads in the failbox will be notified. Failboxes themselves must be robust against asynchronous exceptions, requiring a quite intricate implementation. In Chapter 7, we incrementally developed increasingly more robust failbox implementations. For each implementation we analyzed the vulnerabilities and argued the remedies in the next implementation.

We now investigate whether the vulnerabilities are realistic in the sense that there are tests for them that give a positive result, and also whether the remedies proposed are effective in the sense that these tests turn out negative on the improved version. This is challenging, as the properties to be tested involve subtle coordination issues, requiring instrumentation of Java and also Java Native Interface (JNI) [3] code. We introduce incrementally more powerful tests, guided by the demands of testing incrementally more robust failbox implementations.

As discussed in Chapter 7, the challenge of improving the basic failbox implementation

is to achieve correct behavior of the failbox mechanism in the presence of asynchronous exceptions. Since asynchronous exceptions can occur at any moment during program execution, and hence at any stage of executing the failbox code, both the detection and notification phases of a failbox can be jeopardized. To be able to test the robustness of each failbox implementations in Chapter 7, we need to enforce a scenario in which asynchronous exceptions generating threads are involved and asynchronous exceptions need to be generated in a controlled manner as well.

Our testing approach is based on the delayed execution and the wait-notify patterns as discussed in [98]; we also took inspiration from [5] for the application of countdown latches for synchronization in testing. We extend the wait-notify approach to include handshaking and then apply it to our testing approach, thus improving the precision of the coordination. We also show that the testing approach does not introduce deadlocks as a side effect. The testing approach enables us to develop tests that demonstrate the dependency safety weaknesses of the different failbox implementations, thereby substantiating the earlier claims in Chapter 7. We also consider the generality of the test setup for dependency safety and to what extent tests can be used to provide information about correctness.

This chapter is organized as follows. Section 8.2 introduces the general test setup for testing failboxes on providing dependency safety. Sections 8.3 to 8.7 show how to use delays and latches in tests and tested code to test for dependency safety. Section 8.8 provides an overview of the tests and test results. Section 8.9 contains conclusions and future work.

8.2 Testing Failboxes in the Context of Dependency Safety

In this section we discuss some formal aspects of the test framework that we use in the rest of this chapter. We first give definitions based on [66]. According to the definitions, we provide a test setup that is representative for the dependency safety property in the presence of asynchronous exceptions.

8.2.1 Basic Definitions

We provide some basic definitions for our test setup. We assume that a program state C is defined containing all necessary information about a program execution and the corresponding thread states, e.g., a mapping of program variables (including the program/location counters) to their values, the sequences of statements to be executed by each of the threads, activation records, etc. We assume that the semantics of the language is defined by means of transitions between states of the form $C_k \xrightarrow{t:st} C_{k+1}$, where statement st is executed by a thread with the thread identifier t. An *execution* $E = C_0C_1\ldots$ is a finite or countably infinite sequence of program states. The states in E are indexed by a non-negative integer, and they are referred to as *execution points*. A *thread execution point* is a pair (k, t), where k is an execution point and t is a thread identifier of the thread associated with the execution containing k. Since we assume deterministic threads, any execution of a thread through execution point k leads to the same next execution point $k + 1$. Therefore, we can also refer to k as an execution step taken from C_k.

We assume that a *happens-before* relation, denoted as $\overset{hb_E}{\rightarrow}$, is defined for any given execution E. The happens-before relation is a transitive relation that captures in an intuitive way the causality between operations. For instance, any execution point of a thread happens-before any subsequent execution point of the same thread, i.e., for two execution points of the same thread t, $(k_1, t) \overset{hb_E}{\rightarrow} (k_2, t)$ iff $k_1 < k_2$. Also, assuming $k_1 < k_2$, a thread execution point (k_1, t_1) corresponding to a lock release operation by thread t_1 in state C_{k_1}, happens-before (k_2, t_2) corresponding to an acquire operation on the same lock. Similarly, a (k, t) corresponding to a fork operation by thread t creating a new thread t', happens-before any thread execution point (k', t') of thread t' (where $k < k'$ is implied). The relation is partial in the sense that, e.g., two different locks may be acquired by two different threads at execution points (k_1, t_1) and (k_2, t_2) that are not related by the happens-before relation. If no other dependencies between these execution points exist, there will be executions in which the corresponding operations occur in both orders. For a precise formal definition see [66].

A failing of a thread is modeled by having a special operation **throw**. We say that a thread t fails in an execution point k if an exception is thrown, i.e., the next statement to be executed by t (its continuation) at k is **throw**. Dependency safety is the property that if an operation fails, all other operations depending on its successful completion fail as well. More formally, assuming a given dependency relation D on execution points, we have the following: A program π is *dependency safe* iff for any execution E of π and two thread execution points (k_1, t_1) and (k_2, t_2) of E such that $(k_1, t_1) \overset{hb_E}{\rightarrow} (k_2, t_2)$ and $(k_1, k_2) \in D$ it holds that if t_1 is failing in k_1, then t_2 is failing in k_2.

By different instantiations of the dependency relation D we can obtain different kinds of dependency safety relationships. For instance, by designing the dependency relation such that it expresses dependency between updates of global objects (e.g., variables, arrays, database entries) we can ensure *consistency dependency safety*. It ensures that for each operation B that depends on operation A in the sense that they access the same object, when A fails, B does not operate on an inconsistent state of the object.

Wait dependency safety can be obtained by defining another instance of D. It needs to capture the fact that a wait operation B depends on an operation A since B is blocked waiting for a signal from A. For instance, in Java the case when methods *wait* and *notifyAll* are called on the same object is an example of such a dependency. Wait dependency safety implies for each execution program execution E of a program π the following: if (i) a wait operation B depends on a computation A, and (ii) B terminates if A does not fail, then B terminates.

The consistency dependency safety as well as the wait dependency safety of a program can be ensured using the *failbox* mechanism [63, 66] under the assumption of absence of asynchronous exceptions while executing inside the failbox code. To achieve the consistency dependency safety by using the failbox mechanism, existing synchronizing mechanisms like locks and semaphores in Java need to be adapted to check the state of the failbox [66]. To eliminate the assumption, we provide several failbox implementations in an incremental way in Chapter 7. To test these failbox implementations, we provide a test setup that is representative for the dependency safety property in the presence of asynchronous exceptions. In what follows we only consider wait dependency safety. The test setup can be adapted for checking whether the consistency dependency safety of a program is ensured as well.

8.2.2 Test Setup

Based on the definitions defined in Section 8.2.1, we provide a test setup that is representative for the dependency safety property in the presence of asynchronous exceptions. In particular, we focus on the wait dependency safety of programs.

The setup consists of two threads t_A and t_B that execute operations A and B, respectively. The operations are dependent, i.e., $(A, B) \in D$ and A happens-before B. We refer to this code as the *tested program*. For the test, we also need one thread i which generates asynchronous exceptions. The code corresponding to i we call *test program*. (Later we use two such exception generating threads, but the discussion in the rest of this section applies also to this case without loss of generality.)

Testing in a concurrent setting is challenging because scheduling of threads may influence the outcomes. To enforce a particular testing scenario, i.e., occurrence of asynchronous exceptions in particular program states, synchronization is needed. This is done by preceding the critical operations (locations) with dummy synchronization operations.

For our tests we need to enforce a scenario in which operation A fails, and if as a result operation B fails too, we say that the test is *negative*, otherwise the test is *positive*, in the sense that it was able to demonstrate that the implementation is not correct, i.e., dependency safety does not hold.

To enforce that operation A fails due to an asynchronous exception, we need to execute an operation I triggering such an exception. Furthermore, we want to ensure a happens-before relation between A and I, which may not exist if A and I are performed by different threads. (Note that A and I need not be dependent.) To this end, as shown in Figure 8.1 we instrument the code by replacing operation I by the sequence $W_i; I; S_i$, where W_i and S_i are blocking and unblocking operations, respectively. Also a sequence of operations $S; W$ is added immediately before the operation A. S unblocks W_i whereas W is a blocking operation which can be unblocked only by S_i.

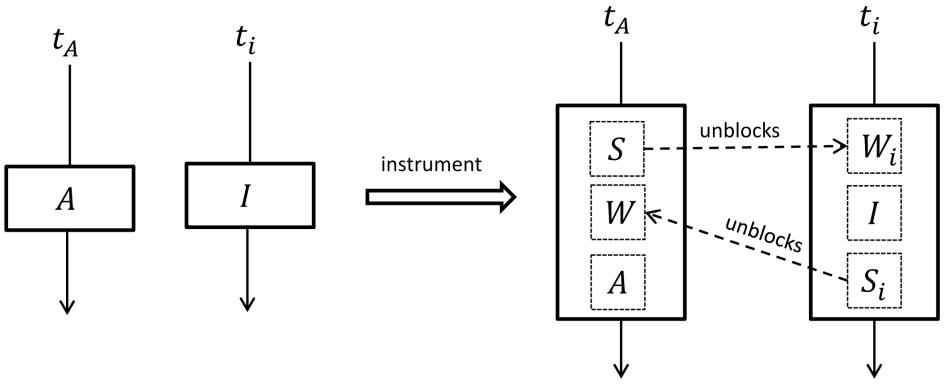

Figure 8.1: Instrumentation

Now, using the definition of happens-before and its transitivity, it is easy to establish for an arbitrary execution E that $S \overset{hb_E}{\to} W_i \overset{hb_E}{\to} I \overset{hb_E}{\to} S_i \overset{hb_E}{\to} W \overset{hb_E}{\to} A$ in Figure 8.2. This is because operation S executed by thread t_A unblocks W_i executed by thread t_i, implying $S \overset{hb_E}{\to} W_i$; operations W_i, I and S_i are executed by thread t_i, implying that $W_i \overset{hb_E}{\to} I \overset{hb_E}{\to} S_i$. Similarly, we can obtain relations $S_i \overset{hb_E}{\to} W$ and $W \overset{hb_E}{\to} A$. Obviously, the above implies I

happens-before A. Moreover, since $S \overset{hb_E}{\rightarrow} I \overset{hb_E}{\rightarrow} A$, we achieve that the execution occurs in E between S and A, corresponding to the location in the original code immediately before A.

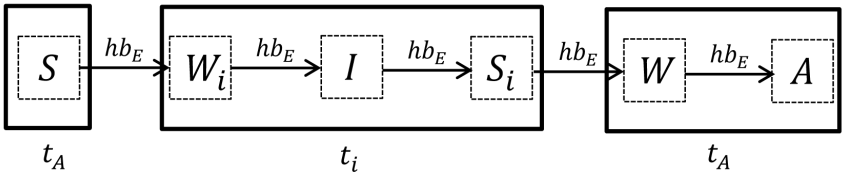

Figure 8.2: The happens-before sequence

Note that the synchronization scheme introduced above does not introduce a deadlock as a side effect. This is because there is never circular waiting between the synchronized threads. From the happens-before sequence established for an arbitrary execution E, it is easy to establish such a sub happens-before relation, i.e., $S \overset{hb_E}{\rightarrow} W_i \overset{hb_E}{\rightarrow} S_i \overset{hb_E}{\rightarrow} W$, as shown in Figure 8.3. Since W does not happen-before S, there is no circular waiting between threads t_A and t_i, implying no deadlock introduced by the synchronization scheme. The above construct can be simplified taking into account that thread t_A is interrupted and stopped while being blocked on W, as in this case W is never executed, and therefore S_i might be omitted.

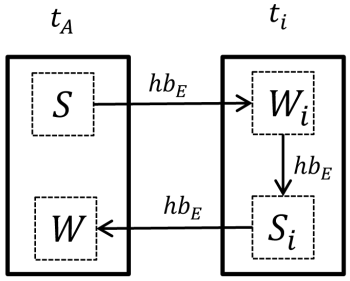

Figure 8.3: Non-circular waiting

We conjecture that our experimental setup is sufficient to ensure that if the implementation passes the test for dependency safety, it will pass that test for any program. We assume that threads t_1 and t_2 run in the same failbox. Threads t_1 and t_2 with the corresponding dependent operations A and B are sufficient abstractions of any thread and dependent operations that can be present in an arbitrary program. So, if the implementation works within our test setup, the dependency safety definition will be satisfied for any set of threads executed in a program.

Based on the test framework presented in this section, we develop tests in a controller manner to demonstrate the wait dependency safety weakness of the different failbox implementations in the rest of this chapter.

From Sections 8.3 to 8.7, the first part of each title indicates how the tests are made more precise by repositioning the occurrence of asynchronous exception inside failbox code. For instance, **Delays within the Try Block** in the title of Section 8.3 indicates that the asynchronous exception occurs during the execution of a thread when it executes

the code inside the Try Block of the corresponding failbox implementation and the synchronization mechanism is implemented by using delays. Similarly, **Latches within the Try Block** in the title of Section 8.4 indicates that the asynchronous exception occurs during the execution of a thread when executing the code inside the Try Block of the failbox implementation as well. However, the synchronization mechanism is implemented by using Latches. Using Latches to synchronize executions of different threads is more robust, as Latches are no longer sensitive to external influences, such as workload on the test machine. As a result, the rest of tests in Sections 8.5, 8.6, and 8.7 are carried out by Latches.

The second part of each title from Sections 8.3 to 8.7 indicates the evolving failbox implementation. For example, in the title of Section 8.3, **NBF** (Non Basic Failbox) indicates the setup do not apply the basic failbox implementation while **BF** (Basic Failbox) represents the basic failbox implementation which is applied to the setup. In a similar way, **UEHF** (Uncaught Exception Handler Failbox) (Java/JNI) in the title of Section 8.6 represents the failbox implementation either in Java or in JNI using the Java uncaught exception handler while **NUEHF** (Non Uncaught Exception Handler Failbox) (JNI) indicates the failbox implementation in JNI without using the Java uncaught exception handler.

8.3 Delays within the Try Block: NBF to BF

In this section, we apply the dependency safety test setup from Section 8.2 using the basic failbox implementation presented in Chapter 7 in the presence of asynchronous exceptions. Before that, we show that the setup without failboxes is not dependency safe. To achieve the required order in which instructions are executed, we use delays.

8.3.1 Testing without failboxes

Test program The interrupting thread i from the abstract setup in Section 8.2 is implemented by the class `InterruptingThread` in Listing 8.1. Operation I is implemented using method `stop` of class `Thread`. This method generates an asynchronous exception in thread t_A (`threadPut`).

Listing 8.1: Class InterruptingThread

```
1 class InterruptingThread extends Thread {
2   Thread interruptedThreadId;
3   InterruptingThread(Thread interruptedThreadId) {
4     this.interruptedThreadId = interruptedThreadId;
5   }
6   @Override
7   public void run() {
8     interruptedThreadId.stop();
9     System.out.println(Thread.currentThread().getName() + "
          stops "+interruptedThreadId.getName());
10  }
11 }
```

Tested program Thread t_A is implemented using class `RunnableBlockPut` in Listing 8.2. The `put` statement (line 10) corresponds to operation A from the setup whose execution needs to be prevented. Instead of the signaling and blocking operations from the abstract test setup we use delays. More precisely, operation W is replaced with `sleep` in line 9. The signaling operation S is simply omitted since we rely on time passage for synchronization.

Listing 8.2: Class RunnableBlockPut with Delay

```
1  class RunnableBlockPut implements Runnable {
2    LinkedBlockingQueue<String> queue;
3    RunnableBlockPut(LinkedBlockingQueue<String> queue) {
4      this.queue = queue;
5    }
6    @Override
7    public void run() {
8      try {
9        Thread.currentThread().sleep(16);
10       queue.put("hello");
11     } catch (InterruptedException e) {
12       //e.printStackTrace();
13     }
14   }
15 }
```

Thread t_B is implemented by class `RunnableBlockTake` in Listing 8.3. Operation B which depends on A (`put`) corresponds to method `take` (line 9).

Listing 8.3: Class RunnableBlockTake

```
1  class RunnableBlockTake implements Runnable {
2    LinkedBlockingQueue<String> queue;
3    RunnableBlockTake(LinkedBlockingQueue<String> queue) {
4      this.queue = queue;
5    }
6    @Override
7    public void run() {
8      try {
9        queue.take();
10     } catch (InterruptedException e) {
11       System.out.println(Thread.currentThread().getName() +
             " is interrputed");
12       //e.printStackTrace();
13     }
14   }
15 }
```

The components of the test and tested program are run from the main program in Listing 8.4. We aim to check if the thread `threadTake` (Listing 8.4 line 6) will be notified when the `threadPut` (Listing 8.4 line 4) crashes before reaching the `put` operation.

Listing 8.4: Class QueueExample without Failbox

```
1 public class QueueExample {
2   public static void main(String[] args) {
3     LinkedBlockingQueue<String> queue = new
        LinkedBlockingQueue<String>();
4     Thread threadPut = new Thread(new RunnableBlockPut(queue
        ));
5     threadPut.start();
6     Thread threadTake = new Thread(new RunnableBlockTake(
        queue));
7     threadTake.start();
8     new InterruptingThread(threadPut).start();
9   }
10 }
```

Output and results To observe sequences of the execution of the program, we add calls of the `println` method at critical places to show its outputs. For instance, a `println` method call is added at line 9 in Listing 8.1 to indicate whether thread `InterruptingThread` sends a "stop" signal to thread `threadPut` and another `println` method call is added at line 11 in Listing 8.3 to indicate whether thread `threadTake` is interrupted. In outputs of all tests demonstrated in this chapter, `Thread-0`, `Thread-1`, and `Thread-2` are strings denoting the names of threads `threadPut`, `threadTake`, and `InterruptingThread`, respectively.

The test produces the following output, as shown in Figure 8.4. If the line `"Thread-1 is interrupted"` is shown as output, we say that the test result is negative; if we do not get this line as output, the test result is positive, i.e., the dependency safety of this program is violated. As output of this test we only get `"Thread-2 stops Thread-0"`, Hence the test result in this section is positive, i.e., the tested program is not dependency safe. In the next section we apply the basic failbox implementation to the tested program and also test whether the dependency safety of the tested program can be ensured via the basic failbox.

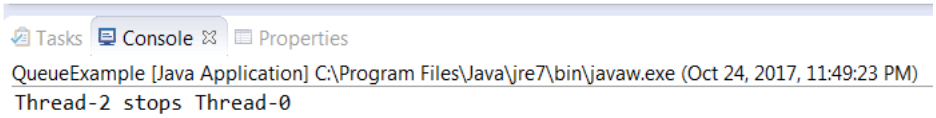

Figure 8.4: The output of the testing without failboxes

Note that we summarize the outputs and results of this and all subsequent tests in Table 8.2. Table 8.1 provides an overview of vulnerability points of different failbox implementations tested through our testing approach by using the synchronization mechanism with Latches which is introduced in Section 8.4.

8.3.2 Testing with basic failbox implementation

To remove the positive test outcome in Section 8.3.1, in an attempt to improve dependency safety, we modify the program by adding the basic implementation of failboxes in Listing 7.3 in Chapter 7. The components of the test setup are modified as follows:

Test program In the new version the interrupting thread (Listing 8.1) remains unchanged.

Tested program The implementations of both threads `threadPut` (Listing 8.2) and `threadTake` (Listing 8.3) remain unchanged.

The class `QueueExample` corresponding to the main program needs to be updated with classes `Failbox` and `FailboxedThread` (as shown in Listing 8.5). A failbox is created (line 3) and is passed to `threadPut` (line 5) and `threadTake` (line 7), respectively. Note that `threadPut` and `threadTake` are instances of class `FailboxedThread`.

Listing 8.5: Class QueueExample with Failbox

```
1 public class QueueExample {
2   public static void main(String[] args) {
3     Failbox failbox = new Failbox();
4     LinkedBlockingQueue<String> queue = new
        LinkedBlockingQueue<String>();
5     Thread threadPut = new FailboxedThread(failbox, new
        RunnableBlockPut(queue));
6     threadPut.start();
7     Thread threadTake = new FailboxedThread(failbox, new
        RunnableBlockTake(queue));
8     threadTake.start();
9     new InterruptingThread(threadPut).start();
10  }
11 }
```

A new class `FailboxedThread` (Listing 8.6) is added to wrap the invocation of the `enter` method of class `Failbox` in Listing 7.3 in Chapter 7. Via the `run` method of class `FailboxedThread`, the two dependent operations, i.e. `put` and `take`, are passed to the `enter` method of the same failbox.

Listing 8.6: Class FailboxedThread with Failbox

```
1 class FailboxedThread extends Thread {
2   Runnable runnable;
3   Failbox failbox;
4   public FailboxedThread(Failbox failbox, Runnable runnable)
        {
5     this.runnable = runnable;
6     this.failbox = failbox;
7   }
8   @Override
9   public void run() {
```

```
10      failbox.enter(this, runnable);
11    }
12  }
```

The instrumented basic failbox implementation is shown in Listing 8.7 with added `println` method calls at lines 3, 21, respectively. The output of the `println` method call at line 3 prints names of threads which enter the failbox. The `println` method call at line 21 indicates whether thread `threadPut` interrupts all other threads running inside the same failbox when thread `InterruptingThread` sends a "stop" signal to thread `threadPut`.

Listing 8.7: The basic implementation – Failbox.enter with println method calls

```
1  public void enter(FailboxedThread thread, Runnable block)
2  {
3    System.out.println(thread.getName() + " enters the failbox
       ");
4    synchronized (this) {
5      if (failed) throw new FailboxException("FailboxFailed");
6      threads.add(thread);
7    }
8    try {
9      try {
10       block.run();
11     } finally {
12       synchronized (this) {
13         threads.remove(thread);
14       }
15     }
16   } catch (Throwable t) {
17     synchronized (this) {
18       failed = true;
19       for (Thread tr : threads) {
20         tr.interrupt();
21         System.out.println(thread.getName() + " interrputs "
               + tr.getName() + " inside the failbox");
22       }
23     }
24     throw t;
25   }
26 }
```

Output and results The test produces the following output, as shown in Figure 8.5.

This output of the program shows that `threadPut` called the interrupt method of `threadTake` inside the failbox after the `InterruptingThread` stopped `threadPut`. Therefore, the result of this test is negative, i.e. the basic implementation of failboxes helps to preserve the dependency safety of the program.

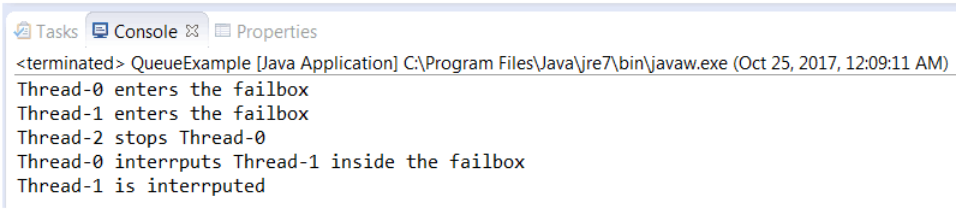

Figure 8.5: The output of the testing with basic failbox implementation

Limitation analysis of the tests Whether or not a desired sequence of executions of the program (in Listings 8.2, 8.3, 8.5 and 8.6) is produced is dependent on many factors. For example, when a machine running this program is under high load, the `stop` method may be called after the delay of 16 milliseconds at line 9 in Listing 8.2 has run out and the `put` has been completed. In that case `threadTake` will finish normally.

8.4 Latches within the Try Block: NBF to BF

To remove the limitations of the test introduced in the previous section, instead of the delays, we use countdown latches in this section to implement the blocking and signaling operations that we referred to in the abstract setup of Section 8.2.

Class `CountDownLatch` from the Java concurrency API is a construct that implements countdown latches. The construct allows one or more threads to wait for a given set of operations to complete. A `CountDownLatch` is initialized with a given count. This count is decremented by calls to the `countDown` method of class `CountDownLatch`. Threads that need to wait for this count to reach zero can do so by calling one of the `await` methods of a `CountDownLatch`. That is, invocation of the `await` method blocks the thread until the count reaches zero.

By using this mechanism, a specific ordering of operations between multiple threads can be forced to occur. This way to achieve a deterministic ordering can make tests more reliable compared to using delays, as it is not sensitive to the current workload of the machine or particular scheduler behavior. Moreover, this approach helps to get fast running tests because the additional synchronisation operations do not take more time than strictly necessary, whereas when using delays, one must estimate how long they actually need to take.

Next, we upgrade the test setup from the previous section using the `CountDownLatch` based synchronization. More precisely, we use two latches `latchA` and `latchB` to this end.

8.4.1 Testing without failboxes

We first test the program without failboxes using Latches instead of delays.

Test program As mentioned above, the blocking and releasing/signaling operations from the abstract setup in Section 8.2 are replaced with operations on countdown Latches.

In particular, we instrument the interrupting thread (Listing 8.8) by adding the call `latchA.await()` corresponding to the W_i operation.

Listing 8.8: Class InterruptingThread with Latch

```
1 class InterruptingThread extends Thread {
2   Thread interruptedThreadId;
3   CountDownLatch latchA;
4   InterruptingThread(Thread interruptedThreadId,
       CountDownLatch latchA) {
5     this.interruptedThreadId = interruptedThreadId;
6     this.latchA = latchA;
7   }
8   @Override
9   public void run() {
10    try {
11      latchA.await();
12    } catch (InterruptedException e) {
13      //e.printStackTrace();
14    }
15    interruptedThreadId.stop();
16    System.out.println(Thread.currentThread().getName() + "
         stops "+interruptedThreadId.getName());
17  }
18 }
```

Tested program In an analogous way, we modify the failing thread `threadPut` (Listing 8.9) by adding the method calls `latchA.countDown()` and `latchB.await()` corresponding to the signaling operation S and the blocking operation W from the abstract setup, respectively. Note that `latchB` is not decreased by any thread.

Listing 8.9: Class RunnableBlockPut with Latch

```
1 class RunnableBlockPut implements Runnable {
2   LinkedBlockingQueue<String> queue;
3   CountDownLatch latchA;
4   CountDownLatch latchB;
5   RunnableBlockPut(LinkedBlockingQueue<String> queue,
       CountDownLatch latchA, CountDownLatch latchB) {
6     this.queue = queue;
7     this.latchA = latchA;
8     this.latchB = latchB;
9   }
10  @Override
11  public void run() {
12    try {
13      latchA.countDown();
14      latchB.await();
15      queue.put("hello");
```

```
16      } catch (InterruptedException e) {
17        //e.printStackTrace();
18      }
19    }
20  }
```

The receiving thread `threadTake` remains unchanged.

The updated class `QueueExample` with Latches but without failboxes is shown in Listing 8.10. Two instances of class `CountDownLatch`, `latchA` and `latchB`, are initialized with a given count 1 repectively at lines 4-5, and passed to the constructor of class `RunnableBlockPut` at line 6. The `latchA` is passed to the constructor of class `InterruptingThread` at line 10.

Listing 8.10: Class QueueExample with Latch

```
1  public class QueueExample {
2    public static void main(String[] args) {
3      LinkedBlockingQueue<String> queue = new
           LinkedBlockingQueue<String>();
4      CountDownLatch latchA = new CountDownLatch(1);
5      CountDownLatch latchB = new CountDownLatch(1);
6      Thread threadPut = new Thread(new RunnableBlockPut(queue
           , latchA, latchB));
7      threadPut.start();
8      Thread threadTake = new Thread(new RunnableBlockTake(
           queue));
9      threadTake.start();
10     new InterruptingThread(threadPut, latchA).start();
11   }
12 }
```

The above mentioned classes implement the abstract setup. The interrupting thread awaits a release signal via the countdown latch from the `put` thread. Immediately before executing the `put` operation the thread decreases the latch, thereby releasing the interrupting thread – and subsequently blocks, awaiting a signal from the interrupting thread. The interrupting thread sends a stop message to the `put` thread which arrives while the `put` thread is blocked. Therefore, `put` is actually never executed. This simple handshaking protocol guarantees that an asynchronous exception always occurs before the `put` operation has completed.

Output and results As expected, the test returns a positive result, i.e., dependency safety does not hold. This is the same result as the one obtained in Section 8.3.1.

8.4.2 Testing with failboxes

Next, we apply the test with Latches on the program with the basic failbox implementation.

Test program The code of the interrupting thread remains the same as in the previous section.

Tested program The two thread implementations remain the same as in the previous section.

As in Section 8.3 we use class `FailboxedThread` to wrap the invocation of the `enter` method of class `Failbox` in Listing 7.3 in Chapter 7. The updated class `QueueExample` with the failbox and Latches is shown in Listing 8.11.

Listing 8.11: Class QueueExample with Failbox and Latches

```
1 public class QueueExample {
2   public static void main(String[] args) {
3     Failbox failbox = new Failbox();
4     LinkedBlockingQueue<String> queue = new
          LinkedBlockingQueue<String>();
5     CountDownLatch latchA = new CountDownLatch(1);
6     CountDownLatch latchB = new CountDownLatch(1);
7     Thread threadPut = new FailboxedThread(failbox, new
          RunnableBlockPut( queue, latchA, latchB));
8     threadPut.start();
9     Thread threadTake = new FailboxedThread(failbox, new
          RunnableBlockTake( queue));
10    threadTake.start();
11    new InterruptingThread(threadPut, latchA).start();
12  }
13 }
```

In this test case, threads `threadPut` and `threadTake` run inside the same failbox, and the synchronization between threads `InterruptingThread` and `threadPut` is implemented by Latches `latchA` and `latchB`, as before.

Output and results As expected, we obtain a negative result, `threadPut` called the interrupt method of `threadTake` inside the failbox after the `InterruptingThread` stopped `threadPut`, i.e. the basic implementation of failboxes helps to preserve the dependency safety of the program. However, in contrast to the result in the previous section, this time, the outcome of the test is more robust, as it is no longer sensitive to external influences, such as the workload on the test machine.

Limitations of the failbox implementation According to the analysis in Chapter 7, the basic implementation of failboxes (Listing 7.3 in Chapter 7) may still crash, i.e., not function as intended, in two cases:

1. If an asynchronous exception happens when `enter` is executed, but before the outer `try` block is entered, the `catch` part, which would notify the other threads, will never be executed.

2. If an asynchronous exception occurs in the `catch`, part of the notification may be interrupted.

In the following sections we will consider incremental improvements of the failbox implementation in Chapter 7 and adjust our tests accordingly to point to the possible vulnerabilities of those implementations.

8.5 Before the Synchronized Statement: BF to UEHF (Java)

In this section, we demonstrate how to adjust the test to point out the vulnerabilities of the basic implementation of failbox. After establishing that those vulnerabilities exist, to remove them we use an improved failbox implementation using an uncaught exception handler in Section 7.3. We show that the adapted test can distinguish between the basic implementation of failbox and the one that uses the uncaught exception handler.

8.5.1 Testing the basic failbox implementation

Test program We take over the implementation of the interrupting thread from Section 8.4 (Listing 8.8).

Tested program The implementations of the threads remain the same, except for the Latch operations, which are now moved to the basic failbox code.

The instrumented basic failbox implementation is shown in Listing 8.12 with the added synchronisation code (lines 3-11 and line 34). If a thread's string name equals to `Thread-0` (line 4), then statements at lines 5-10 will be executed. Here, `Thread-0` is the string name of thread `threadPut` in the previous test case. The synchronization protocol with Latches works as in the previous section. It ensures that an asynchronous exception occurs in `threadPut` before it enters the outer `try` block. In this way, the test execution sequence, revealing the failbox crash case, can be enforced.

Listing 8.12: The basic implementation – Failbox.enter with test code

```
1 public void enter(FailboxedThread thread, Runnable block)
2 {
3   System.out.println(thread.getName() + " enters the failbox
      ");
4   if (thread.getName().equals("Thread-0")) {
5     try {
6       thread.latchA.countDown();
7       thread.latchB.await();
8     } catch (InterruptedException e) {
9       //e.printStackTrace();
10    }
11  }
12  synchronized (this) {
13    if (failed) throw new FailboxException("FailboxFailed");
14    threads.add(thread);
15  }
16  try {
17    try {
18      block.run();
19    } finally {
20      synchronized (this) {
21        threads.remove(thread);
```

```
22        }
23      }
24    } catch (Throwable t) {
25      synchronized (this) {
26        failed = true;
27        for (Thread tr : threads) {
28          tr.interrupt();
29          System.out.println(thread.getName() + " interrputs "
                + tr.getName() + " inside the failbox");
30        }
31      }
32      throw t;
33    }
34  }
```

The class `QueueExample` in this test is similar to the one with the failbox and Latches in Listing 8.11. We only need to pass the `latchA` and `latchB` as the constructor of class `FailboxedThread` instead of the one of class `RunnableBlockPut`, as shown at line 7 in Listing 8.13. This is because the synchronization in Listing 8.9 is moved to inside the `enter` method of class `Failbox` (lines 4-11 in Listing 8.12) to simulate the failing scenario above. The class `FailboxedThread` is updated as shown in Listing 8.14.

Listing 8.13: Positive testing: Class QueueExample with Failbox and Latch

```
1  public class QueueExample {
2    public static void main(String[] args) {
3      Failbox failbox = new Failbox();
4      LinkedBlockingQueue<String> queue = new
          LinkedBlockingQueue<String>();
5      CountDownLatch latchA = new CountDownLatch(1);
6      CountDownLatch latchB = new CountDownLatch(1);
7      Thread threadPut = new FailboxedThread(failbox, new
          RunnableBlockPut(queue), latchA, latchB);
8      threadPut.start();
9      Thread threadTake = new FailboxedThread(failbox, new
          RunnableBlockTake(queue), null, null);
10     threadTake.start();
11     new InterruptingThread(threadPut, latchA).start();
12   }
13 }
```

Listing 8.14: Class FailboxedThread with Failbox and Latch

```
1  class FailboxedThread extends Thread {
2    Runnable runnable;
3    Failbox failbox;
4    CountDownLatch latchA;
5    CountDownLatch latchB;
```

```
6   public FailboxedThread(Failbox failbox, Runnable runnable,
       CountDownLatch latchA, CountDownLatch latchB) {
7     this.runnable = runnable;
8     this.failbox = failbox;
9     this.latchA = latchA;
10    this.latchB = latchB;
11  }
12  @Override
13  public void run() {
14    failbox.enter(this, runnable);
15  }
16 }
```

Output and results In this test, the execution of `threadPut` fails immediately before the synchronized statement inside the basic failbox implementation. The output shows that `threadTake` running inside the same failbox is not notified when `threadPut` fails. So, the result of the test is positive.

Some of the vulnerabilities of the basic failbox implementation can be alleviated by using the mechanism of uncaught exception handlers in Java.

When a thread needs to be terminated because of an uncaught exception, the method `getUncaughtExceptionHandler` of class `Thread` is called on the thread and the `uncaughtException` method is invoked with the thread and the exception as arguments.

In our context, an important feature of the uncaught exception handler is that the method is executed no matter what happens with the thread that has set the handler. This is the last method that is being called before the thread is terminated. We use this feature to provide additional robustness to the failbox implementation. The (instrumented) code of the `enter` method is given in Listing 8.15.

The association between the thread and the failbox, the latter becoming the former's uncaught exception handler, is made in line 3 in Listing 8.15. The part of the previous implementation that notifies all other threads in the failbox is moved to the exception handler in Listing 8.16 via method `uncaughtException` in Listing 8.17. Because now the exceptions can be caught and processed as soon as they happen, the dependence on the vulnerable `try` part is removed.

8.5.2 Testing the failbox implementation with uncaught exception handler

Compared with the basic implementation, the robustness of the implementation of failboxes with uncaught exception handler is improved. So the above test case should also be able to distinguish between the basic implementation of Failboxes and the one that uses the uncaught exception handler.

Test and tested programs All programs remain the same as in the previous section, except for the already mentioned new failbox implementation (Listings 8.15, 8.16, and 8.17). In order to compare these two implementations, we need to choose the fixed location to add the same test code. In this way, the location of the test code does not

influence the distinguishing capability of the test. In the `enter` method of the basic implementation, the test codes (lines 4-11) are added before the synchronized statement at line 12 in Listing 8.12. Therefore, we also add the same test codes at lines 4-11 before the synchronized statement at line 12 in Listing 8.15.

Listing 8.15: The improved test– Failbox.enter method with Uncaught Exception Handler

```
1 public void enter(FailboxedThread thread, Runnable code){
2   System.out.println(thread.getName() + " enters the failbox
       ");
3   thread.setUncaughtExceptionHandler(this);
4   if (thread.getName().equals("Thread-0")) {
5     try {
6       thread.latchA.countDown();
7       thread.latchB.await();
8     } catch (InterruptedException e) {
9       //e.printStackTrace();
10    }
11  }
12  synchronized (this) {
13    if (failed) throw new FailboxException();
14    threads.add(thread);
15  }
16  code.run();
17  thread.setUncaughtExceptionHandler(null);
18 }
```

The call of `println` method is added at line 4 in Listing 8.16 after the `interrupt` method of each thread running inside the failbox is called at line 3.

Listing 8.16: The Failbox.fail method

```
1 public synchronized void fail() {
2   for (Thread tr : threads) {
3     tr.interrupt();
4     System.out.println(tr.getName() + " is interrputed
        inside the failbox");
5   }
6   threads.clear();
7   failed = true;
8 }
```

The implementation of method `uncaughtException` in Listing 8.17 remains the same as in Listing 7.6 in Section 7.3.

Listing 8.17: An Uncaught Exception Handler method

```
1 public void uncaughtException(Thread th, Throwable t) {
2   fail();
3 }
```

Output and results Like the test in the previous Section 8.5.1, the execution of `threadPut` fails at the same vulnerable place, i.e., immediately before the synchronized statement inside the failbox code of the new implementation. However, the output of the test case in this section shows that `threadTake` is interrupted by `threadPut` when it crashes. So the result of the test in this section is negative. Different results in Section 8.5.1 and Section 8.5.2 imply that the new failbox implementation that uses the uncaught exception handler is more robust than the basic one, thereby substantiating the earlier claim in Section 7.3.

Limitation analysis of the failbox implementation As discussed in Section 7.3, also with the new implementation, asynchronous exceptions can still cause problems. The first case in which this can happen is when the exception occurs before the handler is set. The second weak point is when the exception occurs when the handler itself is being executed.

8.6 At the Start of the Enter Method: UEHF (Java) to UEHF (JNI)

In this section, we give a test that produces a positive result with the uncaught exception handler. After that we test an improved version of the uncaught exception handler that is partially implemented in JNI in Section 7.4. The test can also distinguish between these two versions of the uncaught exception handler.

8.6.1 Testing with failbox implementation

Test and tested programs The implementation of the failbox with uncaught exception handler in Java may crash if an asynchronous exception happens in `enter` before the exception handler is set (line 3 in Listing 8.15). The only modification that we need to test this crashing point is to shift the synchronization code from lines 4-9 to line 3 before the exception handler is set. Besides this modification, the rest of the setup is the same as in the previous section.

Output and results The output of the test shows thread `threadTake` running inside the same failbox is not interrupted, i.e., the notification part at lines 2-5 in Listing 8.16 is not executed by `threadPut`.

8.6.2 Testing with improved failbox implementation

The positive test result of the previous section can be remedied by introducing native C code via the Java Native Interface (JNI) to the implementation of the failbox with uncaught exception handler, as shown in Listings 8.18 and 8.19. This implementation is completely analogous to the previous one. In order to keep the implementation of failbox with native C code readable, we wrap all invocations of Java methods as separated C functions. For example, the call of the `getName` method of class `Thread` is wrapped as the function `do_Thread_getName_method` at line 2.

Test and tested programs The only modification that we need to make is to add the same synchronization code (lines 6-9) to the potential crashing point before the exception handler is set at line 11, as shown in Listing 8.18.

Output and results The thread `threadPut` calls the method `Java_Failbox_fail` in Listing 8.19 via the method `uncaughtException` in Listing 8.17 when an asynchronous exception occurs during its execution. That is, `threadTake` is notified by `threadPut` inside the failbox.

Listing 8.18: The Java_Failbox_enter method in C with test code

```
1 JNIEXPORT void JNICALL Java_Failbox_enter(JNIEnv* env,
    jobject failbox, jobject currentThread, jobject body) {
2   jstring name = do_Thread_getName_method(env, currentThread
      );
3   const char* str = (*env)->GetStringUTFChars(env, name,
      NULL);
4   printf("\n %s enters the failbox\n", str);
5   fflush(stdout);
6   if (strcmp(str, "Thread-0") == 0) {
7     do_CountDownLatch_countDown_method(env, currentThread);
8     do_CountDownLatch_await_method(env, currentThread);
9   }
10  (*env)->ReleaseStringUTFChars(env, name, str);
11  do_SetUncaughtExceptionHandler_method(env, currentThread,
      failbox);
12  do_MonitorEnter(env, failbox);
13  jboolean failed_value = do_get_FailboxField_failed(env,
      failbox);
14  if (failed_value == JNI_TRUE) {
15    do_Throw_FailboxException(env, "FailboxFailed");
16  }
17  jobject threads = do_get_FailboxField_threads(env, failbox
      );
18  do_ArrayList_add(env, threads, currentThread);
19  do_MonitorExit(env, failbox);
20  if ((*env)->ExceptionCheck(env) == JNI_TRUE)
21    do_Failbox_fail(env, failbox);
22  do_Runnable_run(env, body);
23  if ((*env)->ExceptionCheck(env) == JNI_FALSE)
24    do_SetUncaughtExceptionHandler_method(env, currentThread
        , NULL);
25 }
```

Like the Java implementation of method `fail` in Listing 8.16, a call of the method `printf` is added to its JNI implementation, as shown at lines 9-11 in Listing 8.19.

Listing 8.19: The Java_Failbox_fail C method

```
1 JNIEXPORT void JNICALL Java_Failbox_fail(JNIEnv* env,
    jobject failbox) {
2   do_MonitorEnter(env, failbox);
3   jobject threads = do_get_FailboxField_threads(env, failbox
      );
4   int arrayListSize = do_ArrayList_size(env, threads);
5   for (int i = 0; i < arrayListSize; i++) {
6     jobject arrayElement = do_ArrayList_get(env, threads, i)
        ;
7     do_Thread_interrupt(env, arrayElement);
8     do_SetUncaughtExceptionHandler_method(env, arrayElement,
        NULL);
9      const char* strName = (*env)->GetStringUTFChars(env,
        nameThread, NULL);
10     printf("\n %s is interrupted inside the faibox\n",
        strName);
11     fflush(stdout);
12  }
13  do_ArrayList_clear(env, threads);
14  do_set_FailboxField_failed(env, failbox, JNI_TRUE);
15  do_MonitorExit(env, failbox);
16 }
```

Limitation analysis of the failbox implementation With the JNI code we addressed only the first vulnerability point. The second weak point, concerning an exception occurring while the handler is being executed, is still there. This is because to call the uncaught exception handler, one needs to briefly go back to Java, i.e., back to the JVM (Listing 8.17). If an asynchronous exception occurs in this method (Listing 8.17) before the native implementation of `fail` in Listing 8.19 has been started (line 2 in Listing 8.17), the remaining threads inside the failbox will not be properly notified.

8.7 At the Start of the Catch Block: UEHF (JNI) to NUEHF (JNI)

In this section we first demonstrate a test revealing the weak spot in the implementation with the uncaught exception handler partially in native code. To remedy the positive result we present a solution fully implemented in JNI in Section 7.5.

8.7.1 Testing with failbox implementation

We modify the test scenario such that thread `threadPut` crashes twice: one crash occurs within the inner `try` block inside the failbox and the failbox is able to catch it; another one happens within the method `uncaughtException` before the `fail` method is executed.

To be able to test these two crashes, we need to add the synchronization approach to the `run` method of class `RunnableBlockPut` (lines 13-14 in Listing 8.9) and the method

uncaughtException (lines 2-15 in Listing 8.19). The updated QueueExample in Listing 8.20 is similar to Listing 8.13. In Listing 8.20, one more InterruptingThread thread (lines 13-14) is created and two more Latches (lines 5-8) are initialized. The first interrupting thread at line 13 stops threadPut when it is waiting at line 15 in Listing 8.9. Then the threadPut crashes and calls uncaughtException method in Listing 8.21. During the execution of the uncaughtException method, threadPut blocks again at line 7 in Listing 8.21 until the second interrupting thread (line 14 in Listing 8.20) calls the stop method of threadPut.

Listing 8.20: Positive test: Class QueueExample with Failbox and Latch

```
1 public class QueueExample {
2   public static void main(String[] args) {
3     Failbox failbox = new Failbox();
4     LinkedBlockingQueue<String> queue = new
        LinkedBlockingQueue<String>();
5     CountDownLatch latchFirstA = new CountDownLatch(1);
6     CountDownLatch latchFirstB = new CountDownLatch(1);
7     CountDownLatch latchSecondA = new CountDownLatch(1);
8     CountDownLatch latchSecondB = new CountDownLatch(1);
9     Thread threadPut = new FailboxedThread(failbox, new
        RunnableBlockPut(queue, latchFirstA, latchFirstB),
        latchSecondA, latchSecondB);
10    threadPut.start();
11    Thread threadTake = new FailboxedThread(failbox, new
        RunnableBlockTake(queue), null, null);
12    threadTake.start();
13    new InterruptingThread(threadPut, latchFirstA).start();
14    new InterruptingThread(threadPut, latchSecondA).start();
15  }
16 }
```

Note that we need to add the call interrupted at line 3 to clear the interrupted flag of the threadPut in Listing 8.21. Otherwise, the threadPut does not block at statement thread.latchSecondB.await(). This is because the interrupted flag is set to true after the first invocation of the method stop of the threadPut has completed.

Listing 8.21: Positive test: An Uncaught Exception Handler method

```
1 public void uncaughtException(Thread t, Throwable e){
2   if (t.getName().equals("Thread-0")) {
3     t.interrupted();
4     try {
5       FailboxedThread thread = (FailboxedThread) t;
6       thread.latchSecondA.countDown();
7       thread.latchSecondB.await();
8     } catch (InterruptedException ee) { }
9   }
10  fail();
11 }
```

The result of the test is positive, i.e., the output shows that `threadTake` is not notified about the failure of `threadPut`.

8.7.2 Testing with improved failbox implementation

The weak spot of this implementation can be resolved by the implementation with fully native code shown in Section 7.5. The test case above should be able to test whether the weak spot still exists in the implementation with fully native code.

To be able to compare the implementations with partially and fully native code, the test code should be added to the same location inside these two different implementations. In the previous case, the test code is added to the start of the handling exceptions part (lines 2-9 in Listing 8.21), i.e., the start of the catch block. In a similar way, the test code needs to be added to the implementation with fully native code, as shown at lines 20-24 in Listing 8.22 when the implementation in fully native code starts handling exceptions.

Listing 8.22: The enter method of Class Failbox with test code in C

```
1 JNIEXPORT void JNICALL Java_Failbox_enter(JNIEnv *env,
    jobject failbox, jobject currentThread, jobject body) {
2   jstring name = do_Thread_getName_method(env, currentThread
      );
3   const char* str = (*env)->GetStringUTFChars(env, name,
      NULL);
4   printf("\n %s enters the failbox\n", str);
5   fflush(stdout);
6   do_MonitorEnter(env, failbox);
7   jboolean failed_value = do_get_FailboxField_failed(env,
      failbox);
8   if (failed_value == JNI_TRUE) {
9     do_Throw_FailboxException(env, "FailboxFailed");
10  }
11  jobject threads = do_get_FailboxField_threads(env, failbox
      );
12  do_ArrayList_add(env, threads, currentThread);
13  do_MonitorExit(env, failbox);
14  do_Runnable_run(env, body);
15  jthrowable exception = (*env)->ExceptionOccurred(env);
16  (*env)->ExceptionClear(env);
17  do_MonitorEnter(env, failbox);
18  do_ArrayList_remove(env, threads, currentThread);
19  if (exception != NULL) {
20    if (strcmp(str, "Thread-0") == 0) {
21      do_Thread_interrupted(env, currentThread);
22      do_CountDownLatch_countDown_method(env, currentThread)
          ;
23      do_CountDownLatch_await_method(env, currentThread);
24    }
25    do_set_FailboxField_failed(env, failbox, JNI_TRUE);
```

```
26      int arrayListSize = do_ArrayList_size(env, threads);
27      for (int i = 0; i < arrayListSize; i++) {
28        jobject arrayElement = do_ArrayList_get(env, threads,
             i);
29        do_Thread_interrupt(env, arrayElement);
30      jstring nameThread = do_Thread_getName_method(env,
           arrayElement);
31      const char* strName = (*env)->GetStringUTFChars(env,
           nameThread, NULL);
32      printf("\n JNI: %s interrupts %s inside the failbox\n",
           str, strName);
33      }
34    }
35    do_MonitorExit(env, failbox);
36    if (exception != NULL)
37      do_Throw(env, exception);
38  }
```

Output and results The output of the test case for the implementation with fully native code shows that `threadTake` is notified by `threadPut`, i.e., the result is negative.

8.8 Overview of Tests and Results

In sections 8.5-8.7, we demonstrate how to test different implementations of failbox using the synchronization mechanism with Latches. By these tests, the improvement of each implementation can be distinguished and also the limitation of each implementation (except for the full one in JNI) assessed. In this section, we show an overview of the tests and their outputs.

In fact, the combined test scenarios now cover all points where the failbox might not handle the exceptions, i.e., by producing negative results, the tests establish the correctness of the final failbox implementation.

Table 8.1 above demonstrates that the robustness of the implementation of failboxes is incrementally improved. For example, the test at row 1 shows that if the test code, i.e., the synchronized mechanism with Latches, is added immediately before the synchronized statement inside each implementation, then the basic one can not handle the asynchronous exception but the one with the uncaught exception handler in Java can. We also use this test case to check the other two implementations, with and without the uncaught exception hander in JNI, which are not given in this chapter. The outputs of the tests show that these two implementations are also able to catch the asynchronous exception. At row 2, the test in Section 8.6 shows that the implementation with uncaught exception handler in Java is not able to catch the asynchronous exception when it occurs at the start of the `enter` method inside the implementation. However, this limitation is removed in the implementation with the uncaught exception handler in JNI, which is shown via the same test in section 8.6.

Table 8.2 provides all results of the tests in this chapter. From the outputs printed on the console, we easily know whether the notification part inside the failbox is executed. In other

Table 8.1: Overview of tests with Latches on implementations of Failboxes

Section	Location of the Test Code inside the failbox	BF (Java)	UEHF (Java)	UEHF (JNI)	NUEHF (JNI)
V	Immediate before the synchronized statement	✗	✓	✓	✓
VI	At the start of the enter method	✗	✗	✓	✓
VII	At the start of the catch block	✗	✗	✗	✓

[1] The checkmarks indicate the implementation is able to catch the asynchronous exception.
[2] The tests for different failbox implementations in gray cells are demonstrated in the corresponding sections of this chapter.
[3] The tests for different failbox implementations corresponding to white cells are not shown in this chapter.

words, whether the implementation is able to handle the asynchronous exception. Note that the abbreviation **T_NBF (Delay)** in the first row represents the test for testing without the basic failbox implementation. Except for this abbreviation, the rest of abbreviations in the first column represent tests for testing with different failbox implementations in Java or in JNI and the required order in which instructions are executed is ensured via either the delay approach or the latches mechanism in corresponding sections. For instance, **T_BF(Java) (Delay)** represents the test for testing the basic Java implementation of failbox and the required order in which instructions are executed is ensured via the delay mechanism in Section 8.3.

8.9 Conclusions and Future Work

We presented a testing approach for concurrent programs that enables to generate asynchronous exceptions in a controlled manner. The approach is motivated, explained and exemplified on implementations of failboxes presented in Chapter 7. The testing approach enables us to develop tests that demonstrate the wait dependency safety weaknesses of the different failbox implementations, thereby substantiating our earlier claims discussed in Chapter 7. Furthermore, the tests are repeatable in that they give the same results for runs that may differ in scheduling, even on different platforms.

A straightforward step for future work is to adapt the test setup for checking whether the consistency dependency safety of programs can be ensured via different failbox implementations in the presence of asynchronous exceptions as well. This will require adaptation of existing synchronizing mechanisms like locks and semaphores in Java by making them aware of failboxes, which is explained in detail in [66]. Another direction to further develop the approach would be to automatically generate synchronization code, and even

Table 8.2: Overview of outputs and results on tests

Section	Output	Result
III (T_NBF) (Delay)	Thread-2 stops Thread-0	Positive
III (T_BF(Java)) (Delay)	Thread-0 enters the failbox Thread-1 enters the failbox Thread-2 stops Thread-0 Thread-0 interrupts Thread-1 inside the failbox Thread-1 is interrupted	Negative
IV (T_NBF) (Latches)	Thread-2 stops Thread-0	Positive
IV (T_BF(Java)) (Latches)	Thread-0 enters the failbox Thread-1 enters the failbox Thread-2 stops Thread-0 Thread-0 interrupts Thread-1 inside the failbox Thread-1 is interrupted	Negative
V (T_BF(Java)) (Latches)	Thread-0 enters the failbox Thread-1 enters the failbox Thread-2 stops Thread-0	Positive
V (T_UEHF(Java)) (Latches)	Thread-0 enters the failbox Thread-1 enters the failbox Thread-2 stops Thread-0 Thread-1 is interrupted inside the failbox Thread-1 is interrupted	Negative
VI (T_UEHF(Java)) (Latches)	Thread-0 enters the failbox Thread-1 enters the failbox Thread-2 stops Thread-0	Positive
VI (T_UEHF(JNI)) (Latches)	JNI: Thread-0 enters the failbox JNI: Thread-1 enters the failbox JNI: Thread-1 is interrupted inside the failbox Thread-2 stops Thread-0 Thread-1 is interrupted	Negative
VII (T_UEHF(JNI)) (Latches)	JNI: Thread-0 enters the failbox JNI: Thread-1 enters the failbox Thread-2 stops Thread-0 Thread-3 stops Thread-0	Positive
VII (T_NUEHF(JNI)) (Latches)	JNI: Thread-0 enters the failbox JNI: Thread-1 enters the failbox JNI: Thread-0 interrupts Thread-1 inside the failbox Thread-2 stops Thread-0 Thread-3 stops Thread-0 Thread-1 is interrupted	Negative

[1] `Thread-0` and `Thread-1` are string names of threads `threadPut` and `threadTake`, respectively. Threads `Thread-2` and `Thread-3` interrupt thread `Thread-0` separately.

the complete tests, for given code with places in that code which are identified as possibly vulnerable for asynchronous exceptions.

In [69], a language and tool are presented to provide test scenarios that, at a rather abstract level, are similar to ours. It would be very interesting to investigate if our tests could be generated by that tool.

Chapter 9

Conclusions

This chapter concludes this thesis by discussing the main contributions and directions for future research. For each of the research questions stated in Chapter 1, we provide the main results and conclusions. Additional details are available in the chapters that cover the research questions.

9.1 Contributions

The main research question covered in this thesis is formulated as follows.

> **RQ:** *How can we ensure the correctness of software that is automatically generated from model-to-code transformations?*

This question is divided into six more specific research questions, and each of these questions is addressed in one of the chapters of this thesis.

The first of these questions concerns the challenges and choices in the implementation and verification of model-to-code transformations and is formulated as follows.

> **RQ$_1$:** *What are the challenges and choices of implementing and verifying Java code generation from concurrent state machines?*

To address this question, in Chapter 3 we identified several challenges and choices in the context of Java code generation from SLCO models which consist of concurrent, communicating objects. The challenges and choices mainly originate in the lack of correspondence between model-oriented primitives in the domain-specific modeling language and their counterparts in Java. Java does not have suitable constructs to directly implement all concepts of SLCO (e.g., atomicity, non-determinism, conditional synchronous and asynchronous communication). To this end, we explored possibilities to mimic each SLCO concept with equivalent observable behavior using appropriate Java constructs. Moreover, high-level modeling languages are not designed to address all details needed when generating executable code from them. Thus, we also worked on the implementation details

which are not defined at the SLCO model level but have to be considered at the Java level (e.g., synchronization constructs and exception handling mechanisms). Furthermore, to obtain good implementations, we took quality aspects (e.g., modularity, efficiency, robustness) into account when implementing SLCO models. However, this consideration of quality aspects imposes additional challenges, e.g., they are often intertwined. Finally, we discussed challenges regarding the verification of the correctness of produced Java code and the transformation itself in terms of a formal logic and proof support.

Once challenges and choices have been identified in the context of implementing SLCO models in Java, gaps between them need to be bridged. This leads to the following research question.

> **RQ_2:** *How to bridge the gaps between DSMLs and their target implementation platforms?*

To address this question, we defined a framework for transforming SLCO models to multi-threaded Java programs in Chapter 4. When implementing different SLCO concepts in Java, we identified and demonstrated concrete gaps between domain-specific modeling languages and envisaged implementation platforms. We also presented how to bridge these gaps by finding patterns in programming languages for correctly capturing concurrent model semantics. We made a distinction between model-generic concepts and model-specific parts of SLCO models and also constructed corresponding Java programs with two parts: generic code and specific code. Generic concepts of SLCO models are transformed into generic Java code, while aspects that are specific for concrete SLCO models are transformed into specific Java code. Quite some effort was put in investigating how to let the generic code be modular. As a beneficial consequence, modification of some constructs of the generic code will not affect the transformation and other constructs of the generic code. Moreover, each construct of the generic code can be understood in isolation which benefits the overall understanding of the transformation. Furthermore, the existing constructs can be reused as much as possible when the need for new target platforms arises.

Once model-to-code transformations have been implemented, a question that naturally arises is how to guarantee that functional properties of the input models are preserved in the generated code. This requires semantic conformance between the input models and the generated code. To address this, a formal specification of the properties and verification of their preservation is needed. Research question RQ_3 addresses this issue.

> **RQ_3:** *How to show that the generic code implementing model-generic concepts preserves certain desirable properties of models?*

In Chapter 5, we first discussed how we had implemented, specified, and verified a protection mechanism to access shared variables in such a way that the code blocks implementing atomic DSML statements are guaranteed to be serializable. This generic mechanism was used in our framework, but the solution can be regarded as a reusable module to safely implement atomic operations in general in concurrent systems. Second, we showed the feasibility to verify the atomicity of generic statements, focusing on the SLCO assignment statement. In particular, we formally proved its implementation against a specification of non-interference using the VeriFast tool [68]. Third, we proved with VeriFast that our mechanism to ensure the atomicity of statements does not introduce *lock-deadlocks*. As an added value, we proved that in our generated programs there is no

need for reentrant locks. This allows us to simplify the formal specification of the Java locks used in our mechanism.

Produced code and the transformations themselves for models in the domain of safety-critical concurrent systems can be complex, which in turn makes a formal proof of their correctness difficult and time-consuming. By systematically structuring a model transformation into small separated and isolated modules, the implementation of each module can be updated and verified without affecting the overall transformation machinery. This in turn requires modular approaches for specifying and verifying each module. Demonstrating the ability to use tool-assisted formal verification for verifying constructs of the generic code in a modular way is therefore important. Thus, we formulated the following research question.

> **RQ_4:** *How to use tool-assisted formal verification to verify in a modular way a construct of the generic code implementing model-independent concepts?*

In Chapter 6, we presented a simplified implementation of the SLCO asynchronous channel as a generic part of the model-to-code transformation from SLCO to Java. We provided its specification and verification in a modular way for use in a general, multi-threaded environment via separation logic in VeriFast. Using VeriFast, we verified the absence of race conditions of the implementation. Besides this, we also showed how to prove properties of clients using the channel. Furthermore, we proposed a novel modular specification schema which improves the modularity of the VeriFast approach. This modular schema was also applied to the specification and verification of the channel. Although our schema is developed using separation logic and VeriFast, it can be straightforwardly adapted for the standard Owicki-Gries method (assuming extensions with modules) or similar formalisms for concurrent verification.

Exceptions are not considered at the SLCO modeling level but have to be handled correctly in the produced code. This is because abnormal termination caused by exceptions may lead to critical issues, such as safety violations and deadlocks. These issues may harm the robustness of the generated Java code. The following research question is related to this.

> **RQ_5:** *How to ensure that generated concurrent code is robust with respect to exceptions?*

To address this question, in Chapter 7 we studied an exception handling mechanism called *failbox*, which is intended to be applied in our current framework. The original implementation of failbox is in Scala, whereas for our setting, we required a Java implementation. Moreover, the original required the assumption of absence of asynchronous exceptions inside the failbox code whereas we wanted to remove that restriction. To address these issues, we first provided a Java implementation without this assumption. The assumption was then eliminated in an incremental manner through several increasingly more robust implementations which are all presented in Chapter 7. For each implementation we analyzed the vulnerabilities and argue the remedies in the next implementation.

Before applying the improved failboxes into our framework for improving the robustness of the generated Java code, the robustness of the mechanism itself needs to be investigated. Research question RQ_6 addresses this issue.

RQ$_6$: *How to assess that the exception mechanism failbox used in the model-to-code transformation is robust?*

In Chapter 8, we presented a testing approach for concurrent programs that enables the generation of asynchronous exceptions in a controlled manner. The approach is motivated, explained and exemplified on implementations of failboxes presented in Chapter 7. The testing approach enables us to develop tests that demonstrate the wait dependency safety weaknesses of the different failbox implementations, thereby substantiating our claims discussed in Chapter 7. Furthermore, the tests are repeatable in that they give the same results for runs that may differ in scheduling, even on different platforms.

Summarizing, techniques and approaches presented in this thesis enable the automated generation of reliable multi-threaded Java programs from SLCO models specifying concurrent systems at a high level of abstraction. We demonstrated that the generic code can be verified using the tool-assisted formal verification in a modular way. Using the VeriFast program verifier we learned that the tool supports Java advanced features like multi-threading, but does not actually provide the specifications for some constructs in Java. For instance, to verify a fine-grained ordered-locking approach proposed in Chapter 5, we needed to supply the specifications of the Java class `ReentrantLock` regarding atomicity and deadlock freedom properties. The specification language of VeriFast is expressive enough to define those specifications by users in a modular manner. However, this required considerable time and expertise.

By making a distinction between model-generic concepts and model-specific parts of SLCO models, we constructed the corresponding Java programs for models with two parts: generic Java code and specific Java code. This distinction provides a clear maintenance advantage and also supports modular verification. In this thesis, we focused on specifying and verifying the constructs of the generic code. This paves the way for verifying the correctness of the transformation completely in the next steps, as specifications for constructs of the generic code can be directly used when verifying the specific Java code. The modularity ensures that each construct of the generic code can be updated without affecting the overall transformation machinery. This makes it possible to have different implementations of each construct in the generic code, e.g., SLCO channels. For example, we only provided the specification and verification of a simplified implementation of SLCO channels. To further investigate the specification and verification for a channel implementation incorporating more advanced concurrency features of Java from package `java.util.concurrent`, an important future extension is to enrich the tool VeriFast by adding more specifications for the concurrent data structures from this package. To improve the robustness of the Java programs generated from models, we studied an exception handling mechanism called failbox. As the original Scala implementation cannot be directly applied to our Java based framework, we provided a Java version, improving it through several increasingly more robust implementations. Specifying and verifying the last and most robust implementation still need to be further investigated. More details about future work are explained in the next section.

9.2 Future Work

A fine-grained generic locking mechanism as demonstrated in Chapter 4 can be regarded as an efficient and reusable module to safely implement atomic operations in concurrent

systems. The mechanism employs a fine-grained ordered-locking approach, in which each shared variable is assigned its own lock for read and write access. The locks of shared variables involved in the Java implementation of an SLCO statement need to be acquired before threads access them. However, not all statements of an SLCO model need to compete over accessing shared variables during the runtime (execution) of the Java program transformed from the SLCO model. In such cases, these statements are data-race free. Correspondingly, acquiring the locks of the involved shared variables are not needed when executing those statements. An interesting direction for future research would be to investigate whether the information about potential data-races can be obtained at the model level. Based on the information obtained, statements that are data-race free would be translated to a Java code block that does not try to acquire locks. Correspondingly, the amount of overhead introduced by using the fine-grained locking is decreased.

Java package `Java.util.concurrent.atomic` contains useful classes to perform atomic operations on single variables. These atomic operations usually are much faster than synchronizing via locks, as they make heavy use of compare-and-swap (CAS). Therefore, another interesting future direction would be to use atomic operations of classes in package `Java.util.concurrent.atomic` to implement the SLCO statements which only involve single shared variables. For instance, `AtomicInteger` and `AtomicBoolean` can be used to implement SLCO Integer and Boolean variables, respectively. To verify the correctness of the corresponding Java implementation of the SLCO statements, specifications for atomic classes are needed. Currently, VeriFast only supports the specification for the atomic class `AtomicReference`. Enriching specifications for atomic classes in Java could lead to an interesting extension to VeriFast. Alternatively, we can investigate whether it is feasible to use other similar verification tools like Vercors [23] to verify the implementation of this kind of SLCO statements. Vercors supports formal specification of classes from the atomic package in the Java API [53] as well.

The fine-grained generic locking mechanism specified and verified in Chapter 5 is simplified in the sense that the wait-notify mechanism for notifying blocked threads is not considered. The wait-notify mechanism improves the efficiency of concurrent programs but also complicates verification. This is because it may cause a liveness issue, i.e., a waiting thread might be blocked forever, leading the program to a deadlock. Also, it restricts certain interleavings in the program. This makes it difficult to reason about the functional behavior of the program. Recently, there has been growing interest in proving deadlock freedom and finite blocking of non-terminating programs [28, 55, 63, 94]. In [55] a useful verification approach for verifying such deadlock freedom of programs that use condition variables is proposed by Hamin et al. This approach could lead to an interesting extension to VeriFast: the extension should enable the verification of the implementation of SLCO statements including the wait-notify mechanism.

A simplified Java implementation of SLCO's asynchronous channel as well as its corresponding specification and verification are demonstrated in Chapter 6. This implementation does not cover the conditional reception feature. Also, the implementation itself does not support blocking send and receive operations. These simplifications facilitate the verification of the implementation. A more complete implementation of SLCO asynchronous channel is presented in Chapter 4, incorporating more advanced features of Java. To verify this improved implementation, we still need to extend the tool VeriFast with specifications of operations of class `BlockingQueue`. A history-based approach to develop behavioral specifications for concurrent data structures like `BlockingQueue` is proposed by Za-

harieva Stojanovski et al. in [94], which provides a valuable insight to formally specify the behavior of Java concurrent data structures for VeriFast in future research.

To improve the robustness of generated code from SLCO models, we implemented and tested an exception handling mechanism called failbox in Chapters 7 and 8, respectively. However, the work presented in this thesis has not yet showed the application of failbox in our framework that transforms SLCO models to Java code. To be able to apply the failbox mechanism in the implementation in future research, one needs to give a definition of all kinds of dependent operations including send and receive operations of channels at the SLCO model level. Based on this definition, an approach to analyze dependent operations of an SLCO model needs to be introduced when transforming from SLCO models to Java code. Correspondingly, dependent operations will be executed in the same failbox during the execution of the Java implementation. Besides applying the failbox concept in the implementation, future work also involves formally verifying the proposed failbox implementations, and comparing the failbox approach to other mechanisms [45, 56] guaranteeing dependency safety. By setting objective criteria, a full evaluation of usability and performance could be performed.

A straightforward step for future work is to adapt the test setup in Chapter 8 for checking whether the consistency dependency safety of programs can be ensured via different failbox implementations in the presence of asynchronous exceptions as well. This will require adaptation of existing synchronizing mechanisms like locks and semaphores in Java by making them aware of failboxes, which is explained in detail in [66]. Another direction to further develop the approach would be to automatically generate synchronization code, and even the complete tests, for given code with places in that code which are identified as possibly vulnerable for asynchronous exceptions. In [69], a language and tool are presented to provide test scenarios that, at a rather abstract level, are similar to ours. It would be very interesting to investigate if our tests could be generated by that tool.

Besides the verification of generic code, specific code should also be verified. This could be addressed by automatically deriving tests for the code from the model [20]. A further step is to generate annotation along with the code. This complementary approach allows fully automated program proofs from the model-to-code generator. A final step would be to directly verify the correctness of the transformation's rules expressed in EGL by using formal approaches. These parts of work can be considered as a valuable direction for future work, as the transformation framework can then be fully verified and thus its correctness can be ensured.

At the SLCO model level, fairness and liveness properties have not been considered yet. However, the challenge when transforming SLCO models to code is that fairness and liveness has to be addressed in some way. The current Java implementation of SLCO models in our framework only ensures fairness properties in the sense that we rely on the constructs supporting fairness provided by the Java library, e.g., fair locks. Moreover, we have achieved and verified lock-deadlock freedom for the generic code. The first step for future research would be to advance the formal rigor of the SLCO-approach by formally expressing fairness and liveness. The second step would be to investigate how to ensure these properties and formally prove their preservation in Java code transformed from SLCO models.

Bibliography

[1] "Eclipse Modeling Framework," https://eclipse.org/modeling/emf/. Accessed: 2017-07-22.

[2] "Java API Specifications," http://docs.oracle.com/javase/7/docs/api/. Accessed: 2017-08-29.

[3] "Java Native Interface Specification," http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html. Accessed 2017-08-11.

[4] "Scala," http://www.scala-lang.org/files/archive/spec/2.11/. Accessed: 2017-07-22.

[5] "Testing Asynchronous Code," https://www.javacodegeeks.com/2015/10/testing-asynchronous-code.html, Accessed: 2016-09-20.

[6] "VeriFast Website," https://people.cs.kuleuven.be/~bart.jacobs/verifast/, Accessed: 2017-06-08.

[7] "Xtext," https://eclipse.org/Xtext/. Accessed: 2017-07-22.

[8] L. Ab Rahim and J. Whittle, "Verifying Semantic Conformance of State Machine-to-Java Code Generators," in *International Conference on Model Driven Engineering Languages and Systems*, ser. LNCS, vol. 6394. Springer, 2010, pp. 166–180.

[9] M. Abadi, C. Flanagan, and S. N. Freund, "Types for Safe Locking: Static Race Detection for Java," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 2, pp. 207–255, 2006.

[10] M. Aigner, A. Biere, C. M. Kirsch, A. Niemetz, and M. Preiner, "Analysis of Portfolio-Style Parallel SAT Solving on Current Multi-Core Architectures," in *Pragmatics of SAT Workshop*, ser. EPiC Series, vol. 29. Easychair, 2013, pp. 28–40.

[11] S. Andova, M. van den Brand, and L. Engelen, "Prototyping the Semantics of a DSL using ASF+SDF: Link to Formal Verification of DSL Models," in *International Workshop on Algebraic Methods in Model-based Software Engineering*, ser. EPTCS, vol. 56, 2011, pp. 65–79.

[12] S. Andova, M. van den Brand, and L. Engelen, "Reusable and Correct Endogenous Model Transformations," in *International Conference on Theory and Practice of Model Transformations*, ser. LNCS, vol. 7307. Springer, 2012, pp. 72–88.

[13] J. Armstrong, "Making Reliable Distributed Systems in the Presence of Software Errors," Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, 2003.

[14] B. J. Arnoldus, "An Illumination of the Template Enigma: Software Code Generation with Templates," Ph.D. dissertation, Eindhoven University of Technology, Eindhoven, The Netherlands, 2010.

[15] J. Arnoldus, M. van den Brand, A. Serebrenik, and J. J. Brunekreef, *Code Generation with Templates*, ser. Atlantis Studies in Computing. Atlantis Press, 2012, vol. 1.

[16] J. Bach, X. Crégut, P. Moreau, and M. Pantel, "Model transformations with Tom," in *International Workshop on Language Descriptions, Tools, and Applications*. ACM, 2012, p. 4.

[17] M. Bagherzadeh, H. Rajan, and M. A. D. Darab, "On Exceptions, Events and Observer Chains," in *Aspect-Oriented Software Development*. ACM, 2013, pp. 185–196.

[18] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of Model Checking*. MIT press, 2008.

[19] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums, "Formal System Development with KIV," in *Fundamental Approaches to Software Engineering*, ser. LNCS, vol. 1783. Springer, 2000, pp. 363–366.

[20] A. Belinfante, J. Feenstra, R. G. de Vries, J. Tretmans, N. Goga, L. M. Feijs, S. Mauw, and L. Heerink, "Formal Test Automation: A Simple Experiment," in *International Workshop on Testing Communicating Systems: Method and Applications*, ser. IFIP, vol. 147. Kluwer, 1999, pp. 179–196.

[21] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond, "DoubleChecker: Efficient Sound and Precise Atomicity Checking," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, pp. 28–39.

[22] J. O. Blech, S. Glesner, and J. Leitner, "Formal Verification of Java Code Generation from UML Models," in *International Fujaba Days 2005-MDD*, 2005, pp. 49–56.

[23] S. Blom and M. Huisman, "The VerCors Tool for Verification of Concurrent Programs," in *International Symposium on Formal Methods*, ser. LNCS, vol. 8442. Springer, 2014, pp. 127–131.

[24] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission Accounting in Separation Logic," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2005, pp. 259–270.

[25] D. Bosnacki, S. Edelkamp, D. Sulewski, and A. Wijs, "GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units," in *International Workshop on Parallel and Distributed Methods in verifiCation*. IEEE Computer Society Press, 2010, pp. 17–19.

[26] D. Bošnački, M. R. Odenbrett, A. Wijs, W. Ligtenberg, and P. Hilbers, "Efficient Reconstruction of Biological Networks via Transitive Reduction on General Purpose Graphics Processors," *BMC Bioinformatics*, vol. 13, p. 281, 2012.

[27] D. Bošnački, M. van den Brand, P. Denissen, C. Huizing, B. Jacobs, R. Kuiper, A. Wijs, M. Wiłkowski, and D. Zhang, "Dependency Safety for Java: Implementing Failboxes," in *International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM, 2016, pp. 15:1–15:6.

[28] P. Boström and P. Müller, "Modular Verification of Finite Blocking in Non-terminating Programs," in *European Conference on Object-Oriented Programming*, ser. LIPIcs, vol. 37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[29] D. Bošnački, M. van den Brand, J. Gabriels, B. Jacobs, R. Kuiper, S. Roede, A. Wijs, and D. Zhang, "Towards Modular Verification of Threaded Concurrent Executable Code Generated from DSL Models," in *International Conference on Formal Aspects of Component Software*, ser. LNCS, vol. 9539. Springer, 2015, pp. 141–160.

[30] S. Brookes, "A Semantics for Concurrent Separation Logic," *Theoretical Computer Science*, vol. 375, no. 1-3, pp. 227–270, 2007.

[31] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2002, pp. 258–269.

[32] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model Checking and the State Explosion Problem," in *Tools for Practical Software Verification, LASER, International Summer School*, ser. LNCS. Springer, 2012, vol. 7682, pp. 1–30.

[33] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A Practical System for Verifying Concurrent C," in *International Conference on Theorem Proving in Higher Order Logics*, ser. LNCS, vol. 5674. Springer, 2009, pp. 23–42.

[34] K. Czarnecki and S. Helsen, "Feature-based Survey of Model Transformation Approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[35] S. de Putter and A. Wijs, "Verifying a Verifier: On the Formal Correctness of an LTS Transformation Verification Technique," in *International Conference on Fundamental Approaches to Software Engineering*, ser. LNCS, vol. 9633. Springer, 2016, pp. 383–400.

[36] S. de Putter and A. Wijs, "Compositional Model Checking Is Lively," in *International Conference on Formal Aspects of Component Software*, ser. LNCS, vol. 10487. Springer, 2017, pp. 117–136.

[37] S. de Putter and A. Wijs, "A Formal Verification Technique for Behavioural Model-To-Model Transformations," *Formal Aspects of Computing*, pp. 1–41, 2017 (available online).

[38] E. Denney and B. Fischer, "Generating Customized Verifiers for Automatically Generated Code," in *International Conference on Generative Programming and Component Engineering*. ACM, 2008, pp. 77–88.

[39] E. Denney, B. Fischer, J. Schumann, and J. Richardson, "Automatic Certification of Kalman Filters for Reliable Code Generation," in *IEEE Aerospace Conference*. IEEE, 2005, pp. 1–10.

[40] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz – Open Source Graph Drawing Tools," in *International Symposium on Graph Drawing*. Springer, 2001, pp. 483–484.

[41] L. Engelen, "From Napkin Sketches to Reliable Software," Ph.D. dissertation, Eindhoven University of Technology, Eindhoven, The Netherlands, 2012.

[42] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," in *ACM Symposium on Operating Systems Principles*. ACM, 2003, pp. 237–252.

[43] A. Farzan and P. Madhusudan, "Causal Atomicity," in *International Conference on Computer Aided Verification*, ser. LNCS, vol. 4144. Springer, 2006, pp. 315–328.

[44] L. M. Feijs, "Transformations of Designs," in *Workshop on Algebraic Methods*, ser. LNCS, vol. 490. Springer, 1991, pp. 167–199.

[45] P. Felber, C. Fetzer, V. Gramoli, D. Harmanci, and M. Nowack, "Safe Exception Handling with Transactional Memory," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001*, ser. LNCS, vol. 8913. Springer, 2015, pp. 245–267.

[46] C. Fetzer, K. Högstedt, and P. Felber, "Automatic Detection and Masking of Non-Atomic Exception Handling," in *International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2003, pp. 445–454.

[47] L. Fix, O. Grumberg, A. Heyman, T. Heyman, and A. Schuster, "Verifying Very Large Industrial Circuits Using 100 Processes and Beyond," *International Journal of Foundations of Computer Science*, vol. 18, no. 1, pp. 45–62, 2007.

[48] C. Flanagan and S. Qadeer, "A Type and Effect System for Atomicity," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2003, pp. 338–349.

[49] C. Fogelberg, A. Potanin, and J. Noble, "Ownership Meets Java," *Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO)*, pp. 30–33, 2007.

[50] M. Fowler, *Domain-specific Languages*. Pearson Education, 2010.

[51] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification*. Java SE 8 Edition, 2015.

[52] D. Grewe and A. Lokhmotov, "Automatically Generating and Tuning GPU Code for Sparse Matrix-Vector Multiplication from a High-Level Representation," in *Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011.

[53] C. Haack, M. Huisman, C. Hurlin, and A. Amighi, "Permission-based Separation Logic for Multithreaded Java Programs," *Logical Methods in Computer Science*, vol. 11, no. 1, pp. 1–66, 2015.

[54] A. Haase, M. Völter, S. Efftinge, and B. Kolb, "Introduction to OpenArchitectureWare 4.1.2," in *MDD Tool Implementers Forum*, 2007.

[55] J. Hamin and B. Jacobs, "Modular Verification of Deadlock Freedom in the Presence of Condition Variables," Department of Computer Science, Katholieke Universiteit Leuven, Tech. Rep., 2017.

[56] D. Harmanci, V. Gramoli, and P. Felber, "Atomic Boxes: Coordinated Exception Handling with Transactional Memory," in *European Conference on Object-Oriented Programming*, ser. LNCS, vol. 6813. Springer, 2011, pp. 634–657.

[57] J. W. Havender, "Avoiding Deadlock in Multitasking Systems," *IBM systems journal*, vol. 7, no. 2, pp. 74–84, 1968.

[58] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[59] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[60] G. J. Holzmann, R. Joshi, and A. Groce, "Tackling Large Verification Problems with the Swarm Tool," in *International SPIN Symposium on Model Checking of Software*, ser. LNCS, vol. 5156. Springer, 2008, pp. 134–143.

[61] K. Huizing and R. Kuiper, "Verification of Object Oriented Programs Using Class Invariants," in *International Conference on Fundamental Approaches to Software Engineering*, ser. LNCS, vol. 1783. Springer, 2000, pp. 208–221.

[62] J. D. Ichbiah, R. Firth, P. N. Hilfinger, O. Roubine, M. Woodger, J.-R. Abrial, J.-L. Gailly, J.-C. Heliard, H. F. Ledgard, B. A. Wichmann *et al.*, *Reference Manual for the Ada Programming Language*. Ada Joint Program Office, 1983.

[63] B. Jacobs, "Provably Live Exception Handling," in *Workshop on Formal Techniques for Java-like Programs*. ACM, 2015, pp. 7:1–7:4.

[64] B. Jacobs, D. Bosnacki, and R. Kuiper, "Modular Termination Verification: Extended Version," Department of Computer Science, Katholieke Universiteit Leuven, Tech. Rep., 2015.

[65] B. Jacobs, P. Müller, and F. Piessens, "Sound Reasoning about Unchecked Exceptions," in *International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, 2007, pp. 113–122.

[66] B. Jacobs and F. Piessens, "Failboxes: Provably Safe Exception Handling," in *European Conference on Object-Oriented Programming*, ser. LNCS. Springer, 2009, vol. 5653, pp. 470–494.

[67] B. Jacobs and F. Piessens, "Expressive Modular Fine-Grained Concurrency Specification," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2011, pp. 271–282.

[68] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java," in *International Symposium on NASA Formal Methods*, vol. 6617. Springer, 2011, pp. 41–55.

[69] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, "Improved Multithreaded Unit Testing," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2011, pp. 223–233.

[70] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[71] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez, *The Epsilon Book*. Eclipse, 2011.

[72] A. Laarman, R. Langerak, J. Van De Pol, M. Weber, and A. Wijs, "Multi-Core Nested Depth-First Search," in *International Symposium on Automated Technology for Verification and Analysis*, ser. LNCS, vol. 6996. Springer, Heidelberg, 2011, pp. 321–335.

[73] A. Laarman, J. van de Pol, and M. Weber, "Multi-core LTSmin: Marrying Modularity and Scalability," in *International Symposium on NASA Formal Methods*, ser. LNCS, vol. 6617. Springer, 2011, pp. 506–511.

[74] G. Lagorio and M. Servetto, "Strong Exception-Safety for Checked and Unchecked Exceptions," *Journal of Object Technology*, vol. 10, no. 1, pp. 1–20, 2011.

[75] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Professional, 2000.

[76] K. R. M. Leino, P. Müller, and J. Smans, "Deadlock-free Channels and Locks," in *European Symposium on Programming*, ser. LNCS, vol. 6012. Springer, 2010, pp. 407–426.

[77] N. D. Matsakis and T. R. Gross, "Handling Errors in Parallel Programs Based on Happens Before Relations," in *IEEE International Symposium on Parallel and Distributed Processing*, 2010, pp. 1–8.

[78] R. Membarth, A. Lokhmotov, and J. Teich, "Generating GPU Code from a High-Level Representation for Image Processing Kernels," in *Workshop on Highly Parallel Processing on a Chip*, ser. LNCS, vol. 7155. Springer, Heidelberg, 2011, pp. 270–280.

[79] M. Musuvathi and D. R. Engler, "Model Checking Large Network Protocol Implementations," in *Symposium on Networked Systems Design and Implementation*. USENIX Association, 2004, pp. 155–168.

[80] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng, "On Optimization Methods for Deep Learning," in *International Conference on Machine Learning*. Omnipress, 2011, pp. 265–272.

[81] P. O'Hearn, J. Reynolds, and H. Yang, "Local Reasoning about Programs that Alter Data Structures," in *International Workshop on Computer Science Logic*, ser. LNCS, vol. 2142. Springer, 2001, pp. 1–19.

[82] S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Communications of the ACM*, vol. 19, no. 5, pp. 279–285, 1976.

[83] S. Pllana and F. Xhafa, *Programming Multicore and Manycore Computing Systems*. Wiley, 2017.

[84] L. A. Rahim and J. Whittle, "A Survey of Approaches for Verifying Model Transformations," *Software & Systems Modeling*, vol. 14, no. 2, pp. 1003–1028, 2015.

[85] H. Rebêlo, R. Coelho, R. Lima, G. T. Leavens, M. Huisman, A. Mota, and F. Castor, "On the Interplay of Exception Handling and Design by Contract: An Aspect-Oriented Recovery Approach," in *Workshop on Formal Techniques for Java-Like Programs*. ACM, 2011, pp. 7:1–7:6.

[86] J. C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *IEEE Symposium on Logic in Computer Science*, IEEE. IEEE Computer Society, 2002, pp. 55–74.

[87] S. Roede, "Proving Correctness of Threaded Parallel Executable Code Generated from Models Described by a Domain Specific Language," Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2012.

[88] D. C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[89] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," *IEEE software*, vol. 20, no. 5, pp. 42–45, 2003.

[90] N. Shavit and D. Touitou, "Software Transactional Memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[91] J. Smans, B. Jacobs, and F. Piessens, "VeriFast for Java: A tutorial," in *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, ser. LNCS. Springer, 2013, vol. 7850, pp. 407–442.

[92] M. Staats and M. P. E. Heimdahl, "Partial Translation Verification for Untrusted Code-Generators," in *International Conference on Formal Engineering Methods*, vol. 5256. Springer, 2008, pp. 226–237.

[93] K. Stenzel, N. Moebius, and W. Reif, "Formal Verification of QVT Transformations for Code Generation," *Software & Systems Modeling*, vol. 14, no. 2, pp. 981–1002, 2015.

[94] M. Z. Stojanovski, "Closer to Reliable Software: Verifying functional Behaviour of Concurrent Programs," Ph.D. dissertation, University of Twente, Enschede, The Netherlands, 2015.

[95] M. Sulzmann and A. Zechner, "Model Checking DSL-Generated C Source Code," in *International SPIN Symposium on Model Checking of Software*, ser. LNCS, vol. 7385. Springer, 2012, pp. 241–247.

[96] K. Svendsen, L. Birkedal, and M. Parkinson, "Joins: A Case Study in Modular Specification of a Concurrent Reentrant Higher-Order Library," in *European Conference on Object-Oriented Programming*, ser. LNCS, vol. 7920. Springer, 2013, pp. 327–351.

[97] K. Svendsen, L. Birkedal, and M. Parkinson, "Modular Reasoning about Separation of Concurrent Data Structures," in *European Symposium on Programming*, ser. LNCS, vol. 7792. Springer, 2013, pp. 169–188.

[98] S. Tasharofi and R. Johnson, "Patterns in Testing Concurrent Programs with Nondeterministic Behaviors," Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep., 2011.

[99] S. Toub, "Keep Your Code Running with the Reliability Features of the .NET Framework," *MSDN Magazine*, October 2015.

[100] T. Tuerk, "A Formalisation of Smallfoot in HOL," in *International Conference on Theorem Proving in Higher Order Logics*, ser. LNCS, vol. 5674. Springer, 2009, pp. 469–484.

[101] M. van Amstel, M. van den Brand, and L. Engelen, "An Exercise in Iterative Domain-specific Language Design," in *In Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*. ACM, 2010, pp. 48–57.

[102] M. van Amstel, M. van den Brand, and L. Engelen, "Using a DSL and Fine-Grained Model Transformations to Explore the Boundaries of Model Verification," in *International Conference on Secure Software Integration and Reliability Improvement*. IEEE Computer Society, 2011, pp. 120–127.

[103] M. F. van Amstel, "Assessing and Improving the Quality of Model Transformations," Ph.D. dissertation, Eindhoven University of Technology, Eindhoven, The Netherlands, 2012.

[104] A. van Deursen, P. Klint, and J. Visser, "Domain-specific Languages: An Annotated Bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[105] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading: Addison-Wesley, 1995.

[106] P. H. Welch and J. M. Martin, "Formal Analysis of Concurrent Java Systems," *Communicating Process Architectures*, vol. 58, pp. 275–301, 2000.

[107] A. Wijs, "Define, Verify, Refine: Correct Composition and Transformation of Concurrent System Semantics," in *International Symposium on Formal Aspects of Component Software*, ser. LNCS, vol. 8348. Springer, 2013, pp. 348–368.

[108] A. Wijs, "GPU Accelerated Strong and Branching Bisimilarity Checking," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 9035. Springer, Heidelberg, 2015, pp. 368–383.

[109] A. Wijs and D. Bošnački, "Many-Core On-The-Fly Model Checking of Safety Properties Using GPUs," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 2, pp. 169–185, 2016.

[110] A. Wijs and L. Engelen, "Efficient Property Preservation Checking of Model Refinements," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 13. Springer, 2013, pp. 565–579.

[111] A. Wijs and L. Engelen, "REFINER: Towards Formal Verification of Model Transformations," in *International Symposium on NASA Formal Methods*, ser. LNCS, vol. 8430, 2014, pp. 258–263.

[112] A. Wijs and B. Lisser, "Distributed Extended Beam Search for Quantitative Model Checking," in *Workshop on Model Checking and Artificial Intelligence*, ser. LNAI, vol. 4428. Springer, Heidelberg, 2007, pp. 165–182.

[113] A. Wijs, T. Neele, and D. Bošnački, "GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking," in *International Conference on Formal Methods*, ser. LNCS, vol. 9995. Springer, 2016, pp. 694–701.

[114] D. Zhang, D. Bošnački, M. van den Brand, P. Denissen, C. Huizing, B. Jacobs, R. Kuiper, A. Wijs, and M. Wiłkowski, "Dependency Safety for Java: Implementing and Testing Failboxes," *Science of Computer Programming*, 2017, submitted.

[115] D. Zhang, D. Bošnački, M. van den Brand, L. Engelen, C. Huizing, R. Kuiper, and A. Wijs, "Towards Verified Java Code Generation from Concurrent State Machines," in *Workshop on Analysis of Model Transformations co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems*, ser. CEUR, vol. 1277. CEUR-WS.org, 2014, pp. 64–69.

[116] D. Zhang, D. Bošnački, M. van den Brand, C. Huizing, B. Jacobs, R. Kuiper, and A. Wijs, "Verifying Atomicity Preservation and Deadlock Freedom of a Generic Shared Variable Mechanism Used in Model-To-Code Transformations," *Communications in Computer and Information Science*, vol. 692, pp. 249–273, 2017.

[117] D. Zhang, D. Bošnački, M. van den Brand, C. Huizing, R. Kuiper, B. Jacobs, and A. Wijs, "Verification of Atomicity Preservation in Model-To-Code Transformations," in *International Conference on Model-Driven Engineering and Software Development*. SciTePress, 2016, pp. 578–588.

[118] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst, "Ownership and Immutability in Generic Java," in *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2010, pp. 598–617.

[119] A. Zündorf, "Rigorous Object Oriented Software Development," Ph.D. dissertation, University of Paderborn, Paderborn, Germany, 2001.

Summary

From Concurrent State Machines to Reliable Multi-threaded Java Code

Model transformation is a powerful concept in model-driven software engineering. Starting with an initial model written in a domain-specific modeling language (DSML), other artifacts such as additional models, source code and test scripts can be produced via a chain of transformations. The initial model is typically written at a conveniently high level of abstraction, allowing the user to reason about complex system behavior in an intuitive way. The model transformations are supposed to preserve the correctness of the initial model, thereby realizing a framework where the generated artifacts are correct by construction. A question that naturally arises for model-to-code transformations is how to guarantee that the functional properties of the input models are preserved in the generated code. We distinguish generic and model specific code and concentrate on the former. We consider this question for a framework that implements the transformation from the concurrent DSML SLCO to multi-threaded Java code. In this context, we identify and address several challenges involving the robustness and correctness of generated code.

Our main contributions are as follows:

First, we start our research by studying the challenges and choices of implementing and verifying model-to-code transformations. We explore possibilities to mimic each SLCO concept with a counterpart programmed in Java. We also work on the implementation details which are not defined at the SLCO model level but have to be considered at the Java level (e.g., synchronization constructs and exception handling mechanisms). Furthermore, to obtain good implementations, we take quality aspects (e.g., modularity, efficiency, robustness) into account when implementing SLCO models. This consideration of quality aspects imposes additional challenges, e.g., functional and quality issues are often intertwined. Finally, we discuss challenges regarding the verification of the correctness of produced Java code and the transformation itself in terms of a formal logic and proof support.

Second, we develop an automated model-to-code transformation from SLCO models to multi-threaded Java programs, which is implemented in the Epsilon Generation Language (EGL) using Eclipse. The transformation rules are defined by means of templates. The generator applies transformation rules to all the meta-model objects, which results in generation of the corresponding Java code. This Java code is constructed by combining specific code implementing the behavior of the input model with generic code implementing model independent SLCO concepts. By making a distinction between generic and specific

code, proving the correctness of model-to-code transformations can be done more efficiently.

Third, we provide a protection mechanism to access shared variables to preserve atomicity of SLCO statements in the Java implementation. This generic mechanism is used in our framework, but the solution is reusable to safely implement atomic operations in concurrent systems. We give its generic specification based on separation logic and verify it using VeriFast. We also show the feasibility to verify the atomicity of statements, focusing on the SLCO assignment statement. Moreover, we prove with VeriFast that our mechanism does not introduce lock-deadlocks. As an added value, we prove that in our generated programs there is no need for reentrant locks, which allows us to simplify the formal specification of the Java locks in our mechanism.

Fourth, we specify and perform automated verification of a Java implementation of the SLCO channel data type. To this end, we introduce a new schema that supports fine grained concurrency and procedure-modularity. We demonstrate this approach by specifying and verifying the channel construct using the separation logic based tool VeriFast. Our results show that such tool-assisted formal verification is a viable technique, supporting object orientation, concurrency via threads, and parameterized verification.

Fifth, we ensure the robustness of generated code by applying the exception handling mechanism called failbox to the code generation. To this end, we implement the mechanism failbox in Java and also improve it by eliminating the assumption required in the original failbox implementation. This assumption is eliminated in an incremental manner through several increasingly more robust implementations. For each implementation we analyze the vulnerabilities and argue the remedies in the next implementation.

Last, we present a testing approach to investigate whether the vulnerabilities of each implementation of failbox are realistic and the remedies proposed in the next implementation are effective. This testing approach enables us to generate asynchronous exceptions in a controlled manner for concurrent programs. The tests are repeatable in that they give the same results for runs that may differ in scheduling, even on different platforms.

To summarize, we identify several challenges on the correctness and robustness of Java code generated from concurrent state machines. To address these challenges, we introduce an approach where code is verified in a modular fashion using VeriFast and robustness is provided by an improved failbox mechanism. Techniques and approaches presented in this thesis enable the automated generation of reliable multi-threaded Java programs from SLCO models specifying concurrent systems at a high level of abstraction.

Curriculum Vitae

Personal Information

Name: Dan Zhang
Date of birth: June 30, 1986
Place of birth: Henan, China

Education

MSc. in Computer Science and Engineering *2010–2013*
Xi'an Jiaotong University
Xi'an, China

BSc. in Computer Science and Engineering *2005–2009*
Zhengzhou University of Light Industry
Zhengzhou, China

Professional Experience

Researcher *2017–present*
University of Twente
Enschede, the Netherlands

PhD candidate *2013–2017*
Eindhoven University of Technology
Eindhoven, the Netherlands

Titles in the IPA Dissertation Series since 2015

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World.* Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction.* Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems.* Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness.* Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle.* Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking.* Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography.* Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems.* Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars.* Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology.* Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction.* Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things.* Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes.* Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05