Report from Dagstuhl Seminar 22451

Principles of Contract Languages

Dilian Gurov[*1], Reiner Hähnle[*2], Marieke Huisman[*3], Giles Reger[*4], and Christian Lidström[†5]

1 **KTH Royal Institute of Technology – Stockholm, SE.** dilian@kth.se
2 **TU Darmstadt, DE.** haehnle@cs.tu-darmstadt.de
3 **University of Twente – Enschede, NL.** m.huisman@utwente.nl
4 **University of Manchester, GB.** giles.reger@manchester.ac.uk
5 **KTH Royal Institute of Technology – Stockholm, SE.** clid@kth.se

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 22451 "Principles of Contract Languages". At the seminar, participants discussed the fundamental aspects of software contracts. Topics included the format and expressiveness of contracts, their use cases in software development and analysis, and contract composition and decomposition.

Seminar November 6–11, 2022 – http://www.dagstuhl.de/22451
2012 ACM Subject Classification Theory of computation → Program specifications
Keywords and phrases software contracts, program specifications, software development, program analysis
Digital Object Identifier 10.4230/DagRep.12.11.1

1 Executive Summary

Dilian Guro
Reiner Hähnle
Marieke Huisman
Giles Reger

This report documents the program and the outcomes of Dagstuhl Seminar 22451 "Principles of Contract Languages".

Formal, precise analysis of non-trivial software is a task that necessarily must be decomposed. The arguably most important composition principle in programming is the procedure (function, method, routine) call. For this reason, it is natural to decompose the analysis of a program along its call structure. Decomposition in this context means to replace a procedure call with a declarative description, possibly an approximation, of the call's effect. In his seminal work on runtime verification in Eiffel, Bertrand Meyer suggested to use the metaphor of a contract between the user (caller) and implementor (callee) for such a description.

Contracts continue to be a central element in run-time (dynamic) analysis. In the last two decades they also became the dominant decomposition approach in deductive verification and are realized in all major software verification systems. More recently, software contracts are increasingly used in test case generation and model checking. Furthermore, programming

[*] Editor / Organizer
[†] Editorial Assistant / Collector

languages such as “Racket” or “Dafny” were designed with a notion of contract. Contract-based specification languages are available for mainstream programming languages, notably JML for “Java” and ACSL/ACSL++ for “C”/“C++”.

However, there is considerable fragmentation concerning terminology, basic principles, expressivity, and usage of contracts in different research communities. Therefore, this Dagstuhl Seminar convened researchers working with contracts in static verification, runtime verification, as well as testing, with the goal of creating a unified view on software contracts.

The seminar participants discussed the following topics and questions:

1. *Sub-procedural* contracts: contracts for blocks, loops, suspension points, barriers
2. Combining *trace-based specifications* for *global* properties with *two-state* contracts for recursive procedures
3. Rethink *abstract* versus implementation-layer specifications
4. Contracts for *concurrent languages*
5. Contract *composition*
6. Unify contracts for deductive and runtime verification and testing
7. (Behavioral) *types* as lightweight contracts
8. Where do contracts come from? Contract *synthesis*
9. Contract *validation*, connection to *natural language*
10. Contracts for *refinement* (correctness by construction)
11. Contracts for relational properties
12. Debugging contracts
13. Domain-specific contract languages

2 Table of Contents

Executive Summary
Dilian Gurov, Reiner Hähnle, Marieke Huisman, and Giles Reger 1

Overview of Talks

Industrial Experience with Specification
David Cok 5

Executable Contracts in Ada and SPARK
Claire Dross 5

Loop Verification with Invariants and Contracts
Gidon Ernst 5

Explaining Contract Violations with BeepBeep 3
Sylvain Hallé 6

Contracts in Model Checking
Paula Herber 6

A Program Logic for Data Dependence Analysis
Asmae Heydari Tabar 7

Formal Specification with Contracts: Past, Present, Future
Reiner Hähnle 7

A Contract-based View on Functional Equivalence Checking
Marie-Christine Jakobs 8

Towards Domain-Specific Contracts for Programs
Eduard Kamburjan 8

Formal Verification of a JavaCard Virtual Machine for Common Criteria Certification
Nikolai Kosmatov 9

From English Assume-Guarantee Contracts to Validated Temporal Logic Specifications
Kristin Yvonne Rozier 9

Efficient First-Order Runtime Verification
Srdan Krstic 10

Formal Specifications Investigated: A Classification and Analysis of Annotations for Deductive Verifiers
Sophie Lathouwers 10

Contracts, abstractly
Christian Lidström 11

Contracts in Static Verification
Rosemary Monahan 11

On Automatic Contract Synthesis
Philipp Rümmer 12

When You Hit the Frame
Thomas Santen 13

Correctness-by-Construction: The CorC Ecosystem
Ina Schaefer 13

VeyMont: Parallelising Verified Programs instead of Verifying Parallel Programs
Petra van den Bos 13

Working groups

Two-state versus Trace-based Contracts: Report from Breakout Group 1
Dilian Gurov, Bernhard Beckert, Alessandro Cimatti, Paula Herber, Srdan Krstic, Martin Leucker, Christian Lidström, and Marco Scaletta 14

Abstraction in Contracts: Report from Breakout Group 2
Reiner Hähnle, Asmae Heydari Tabar, Eduard Kamburjan, Nikolai Kosmatov, Rosemary Monahan, Thomas Santen, Ina Schaefer, Julien Signoles, and Alexander J. Summers 18

Specification Engineering: Report from Breakout Group 3
Wolfgang Ahrendt, David Cok, Gidon Ernst, Sophie Lathouwers, Giles Reger, Kristin Yvonne Rozier, and Philipp Rümmer 21

Interoperable Contracts: Report from Breakout Group 4
Petra van den Bos, Claire Dross, Sylvain Hallé, Marieke Huisman, Marie-Christine Jakobs, and Mattias Ulbrich 24

Participants 27

3 Overview of Talks

3.1 Industrial Experience with Specification

David Cok (Safer Software Consulting – Rochester, US)

Academic and workshop verification problems tend to focus on proving easily described and compact algorithms. When specifying and verifying legacy industrial code, however, such algorithmic verification is not the main challenge. Rather other aspects of DV are important: validating that formal specifications correspond to the real requirements, handling of frame conditions, both reads and modifies conditions, in large bodies of code; effective specification and verification combined with abstraction; and the ability to efficiently debug failed proofs.

3.2 Executable Contracts in Ada and SPARK

Claire Dross (AdaCore – Paris, FR)

This talk presents the contracts available in the Ada language. They have an executable semantics and can be checked both dynamically and statically through the deductive verification tool SPARK. The talk mostly focuses on the challenges that we are currently facing both for dynamic analysis (generation of test values which comply with the precondition, coverage of the postcondition...) and for deductive verification (support of non-executable constructs such as quantification or logical equality, contracts for pointer support and ownership...).

3.3 Loop Verification with Invariants and Contracts

Gidon Ernst (LMU München, DE)

Main reference Gidon Ernst: "Loop Verification with Invariants and Contracts", in Proc. of the Verification, Model Checking, and Abstract Interpretation – 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings, Lecture Notes in Computer Science, Vol. 13182, pp. 69–92, Springer, 2022.
URL https://doi.org/10.1007/978-3-030-94583-1_4

Invariants are the predominant approach to verify the correctness of loops. As an alternative, loop contracts, which make explicit the premise and conclusion of the underlying induction proof, can sometimes capture correctness conditions more naturally. But despite this advantage, the second approach receives little attention overall. In this talk we explore the fundamentals of loop contracts in an accessible way, demonstrate constructive translations from and to purely invariant-based proofs, characterize completeness of both approaches, and discuss their relative strengths and weaknesses on well-known verification problems, cf. [1]. Finally, we point out a few ideas on how loop contracts can perhaps lead to novel ways of automating correctness proofs of loopy programs.

References

1 Gidon Ernst. *Loop Verification with Invariants and Contracts*. In Proc. of VMCAI, Springer, LNCS, 2022.

3.4 Explaining Contract Violations with BeepBeep 3

Sylvain Hallé (University of Quebec at Chicoutimi, CA)

Main reference Sylvain Hallé: "Explainable Queries over Event Logs", in Proc. of the 24th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2020, Eindhoven, The Netherlands, October 5-8, 2020, pp. 171–180, IEEE, 2020.
URL https://doi.org/10.1109/EDOC49727.2020.00029

It is one thing to discover that the conditions stipulated by a contract have been violated –but it is another thing to provide an explanation for this violation. And what is an explanation anyway? In this short talk, we explored the notion of explanation for specifications expressed on event streams, and showed how an existing event stream processing engine called BeepBeep has been retrofitted to automatically compute explanations for the violation of a specification.

3.5 Contracts in Model Checking

Paula Herber (Universität Münster, DE)

Joint work of Paula Herber, Timm Liebrenz, Pauline Blohm
Main reference Paula Herber, Timm Liebrenz: "Dependence Analysis and Automated Partitioning for Scalable Formal Analysis of SystemC Designs", in Proc. of the 18th ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2020, Jaipur, India, December 2-4, 2020, pp. 1–6, IEEE, 2020.
URL https://doi.org/10.1109/MEMOCODE51338.2020.9314998

In this talk, I have started with a brief history of model checking, introduced the general idea, and sketched some major achievements that have significantly increased the feasibility of model checking over the years. Then, I have presented two examples from our own work where we have applied model checking to embedded systems: An approach where we proposed automated partitioning of hardware/software co-designs and then model checked the partitions, and another approach where we proposed to use contracts for the safe integration of reinforcement learning into embedded control systems. I have discussed some ideas we had for modular verification that actually enabled us to verify complex systems, but I also demonstrated where we failed and where I think introducing a notion of contracts into the verification process would be beneficial, if good abstractions and constructs to express these abstraction were available.

References

1 Edmund M., Clarke Jr., Orna Grumberg, Daniel Kroening, Doron Peled and Helmut Veith. *Model Checking. Second Edition*. MIT 2018.
2 Paula Herber, Timm Liebrenz. *Dependence Analysis and Automated Partitioning for Scalable Formal Analysis of SystemC Designs*. In Proceedings of the ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), 2020.

3.6 A Program Logic for Data Dependence Analysis

Asmae Heydari Tabar (TU Darmstadt, DE)

Joint work of Asmae Heydari Tabar, Richard Bubel, Reiner Hähnle
Main reference Asmae Heydari Tabar, Richard Bubel, Reiner Hähnle: "Automatic Loop Invariant Generation for Data Dependence Analysis", in Proc. of the 10th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2022, Pittsburgh, PA, USA, May 22-23, 2022, pp. 34–45, ACM, 2022.
URL https://doi.org/10.1145/3524482.3527649

Parallelization of programs relies on sound and precise analysis of data dependences in the code, specifically, when dealing with loops. State-of-art tools are based on dynamic profiling and static analysis. They tend to over- and, occasionally, to under-approximate dependences. The former misses parallelization opportunities, the latter can change the behavior of the parallelized program. We have developed a sound and highly precise approach to generate data dependences based on deductive verification. The central technique is to infer a specific form of loop invariant tailored to express dependences. To achieve full automation, we adapt predicate abstraction in a suitable manner. To retain as much precision as possible, we generalized logic-based symbolic execution to compute abstract dependence predicates. We implemented our approach for Java on top of a deductive verification tool. The evaluation shows that our approach can generate highly precise data dependences for representative code taken from HPC applications.

3.7 Formal Specification with Contracts: Past, Present, Future

Reiner Hähnle (TU Darmstadt, DE)

Contract-based formal specification languages are popular in deductive verification, runtime verification and, increasingly, in model checking and test case generation. We outline the building blocks of contract languages and trace some historical developments, as well as current usage. Finally, we suggest some research challenges:

1. *Sub-procedural* contracts
   - blocks, loops, suspension points, barriers
2. Combine *trace-based specification* for *global* properties with *lossless* contracts for recursive procedures
3. Rethink *abstract* vs. implementation-layer specs
4. Contracts for *concurrent languages*
5. Contract *composition*
6. Unify contracts for deductive & runtime verification and testing
7. (Behavioral) *types* as lightweight contracts
8. Where do contracts come from? Contract *synthesis*
9. Contract *validation*, connect to *natural language*
10. Contracts for *refinement* (correctness by construction)
11. Contracts for relational properties
12. Debugging contracts
13. Domain-specific contract languages

3.8 A Contract-based View on Functional Equivalence Checking

Marie-Christine Jakobs (TU Darmstadt, DE)

Main reference Marie-Christine Jakobs: "PEQCHECK: Localized and Context-aware Checking of Functional Equivalence", in Proc. of the 9th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2021, Madrid, Spain, May 17-21, 2021, pp. 130–140, IEEE, 2021.
URL https://doi.org/10.1109/FormaliSE52586.2021.00019

So far, I have used software verifiers to check functional equivalence of two programs, the original and refactored one. To this end, I split the check for functional equivalence into one check for each refactored code segment [1] (code segments are identified by the user). Each check of a refactored code segment inspects the functional equivalence of original and refactored code segment. Therefore, I encode the check of each refactored code segement into a separate verification task. In this talk, I will look into an alternative to describe the verification task, namely contracts. This may allow us to use other tools, e.g., deductive verifiers to check the equivalence of the two code segments.

References

1 M.-C. Jakobs. PEQcheck: Localized and context-aware checking of functional equivalence. In *Proc. FormaliSE*, pages 130–140. IEEE, 2021.

3.9 Towards Domain-Specific Contracts for Programs

Eduard Kamburjan (University of Oslo, NO)

Semantically lIfted programs [1] enable programs to explicitly refer to their own program state as a *knowledge graph*, and add information about the application domain through the integration of ontologies. This talk discussed the challenges and chances for contracts in this setting: using ontologies, we can formulate contracts using domain knowledge *together* with the domain expert, yet stay in a fully formal setting thanks to the rich theories developed the semantic web. However, the questions how exactly to (a) connect with these theories, and (b) how to use languages for data access, such as SPARQL, in a contract language beyond simple typing guarantees [2] remain open.

References

1 Kamburjan, E., Klungre, V., Schlatte, R., Johnsen, E. & Giese, M. Programming and Debugging with Semantically Lifted States. *ESWC*. (2021)
2 Kamburjan, E. & Kostylev, E. Type Checking Semantically Lifted Programs via Query Containment under Entailment Regimes. *Description Logics*. (2021)

3.10 Formal Verification of a JavaCard Virtual Machine for Common Criteria Certification

Nikolai Kosmatov (Thales Research & Technology – Palaiseau, FR)

Joint work of Adel Djoudi, Martin Hána, Nikolai Kosmatov
Main reference Adel Djoudi, Martin Hána, Nikolai Kosmatov: "Formal Verification of a JavaCard Virtual Machine with Frama-C", in Proc. of the Formal Methods – 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings, Lecture Notes in Computer Science, Vol. 13047, pp. 427–444, Springer, 2021.
URL https://doi.org/10.1007/978-3-030-90870-6_23

Formal verification of real-life industrial software remains a challenging task. It provides strong guarantees of correctness, which are particularly important for security-critical products, such as smart cards. Security of a smart card strongly relies on the requirement that the underlying JavaCard virtual machine ensures necessary isolation properties. This talk presents a recent formal verification of a JavaCard Virtual Machine implementation performed by Thales for Common Criteria certification using the Frama-C verification toolset. The certification was successful: an EAL6 certificate was delivered in October 2021 and an EAL7 certificate (the highest one) was delivered in October 2022. This is the first verification project for such a large-scale industrial smart card product where deductive verification is applied on the real-life C code. The target properties include common security properties such as integrity and confidentiality. The implementation contains over 7,000 lines of C code. After a formal specification in the ACSL specification language, over 50,000 verification conditions were generated and successfully proved. We give an overview of the project and proof results, and focus on the mataproperty based approach that was a key solution for a successful specification and verification of security properties. This is a joint work with Adel Djoudi and Martin Hána.

3.11 From English Assume-Guarantee Contracts to Validated Temporal Logic Specifications

Kristin Yvonne Rozier

Major safety-critical systems, like the NASA Lunar Gateway Vehicle System Manager, utilize structured English requirements (called contracts) with strict requirements for validation and traceability from early design time through system runtime. We overview the successes and challenges of this approach, highlight recent work in automated temporal logic contract validation, and point out the most immediate needs for future research.

3.12 Efficient First-Order Runtime Verification

Srdan Krstic (ETH Zürich, CH)

Joint work of Martin Raszyk, David A. Basin, Srdan Krstic, Dmitriy Traytel
Main reference Martin Raszyk, David A. Basin, Srdan Krstic, Dmitriy Traytel: "Practical Relational Calculus Query Evaluation", in Proc. of the 25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference), LIPIcs, Vol. 220, pp. 11:1–11:21, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
URL https://doi.org/10.4230/LIPIcs.ICDT.2022.11

Existing arbitrary first-order (FO) formula evaluation approaches are less efficient than the established algorithms based on finite tables which are used in existing database management systems. However, latter cannot handle arbitrary FO formulas, but rather those that have a particular syntactic structure, e.g., isomorphic to relational algebra expressions.

I will showcase a new translation from arbitrary FO formula into two relational algebra expressions for which the finiteness of the formula's evaluation result is guaranteed. Assuming an infinite domain, the two expressions have the following meaning: The first is closed and characterizes the original formula's relative safety, i.e., whether given a fixed database, the original formula evaluates to a finite relation. The second expression is equivalent to the original formula, if it is relatively safe. The translation improves the time complexity over existing approaches, which we also empirically confirm in both realistic and synthetic experiments.

3.13 Formal Specifications Investigated: A Classification and Analysis of Annotations for Deductive Verifiers

Sophie Lathouwers (University of Twente – Enschede, NL)

Joint work of Sophie Lathouwers, Marieke Huisman
Main reference Sophie Lathouwers, Marieke Huisman: "Formal Specifications Investigated: A Classification and Analysis of Annotations for Deductive Verifiers", in Proc. of the 10th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2022, Pittsburgh, PA, USA, May 22-23, 2022, pp. 69–79, ACM, 2022.
URL https://doi.org/10.1145/3524482.3527652

This talk investigates what annotations are actually needed for deductive verification, and provides a taxonomy to categorise these annotations. In particular, we identify several top-level categories, which are further divided into subcategories of annotations. This taxonomy is then used as a basis to investigate how often particular annotation categories occur, by inspecting over 10k lines of annotated programs. To determine whether the results are in line with expectations, we have interviewed several experts on deductive verification. Moreover, we show how the results can be used to evaluate the effectiveness of annotation generators. The knowledge from this analysis provides a gateway to guide further research in improving the efficiency of deductive verification, e.g.: it can serve as a guideline on what categories of annotations should be generated automatically, to evaluate the power of existing annotation generation techniques, and to improve the teaching of deductive verification.

3.14 Contracts, abstractly

Christian Lidström (KTH Royal Institute of Technology – Stockholm, SE)

In our view, the primary purpose of contracts is to split responsibilities, and assign them to different parts of a system. To this end, we propose an abstract view of contracts as the formal foundation. Treating contracts as mathematical objects enable elegant formalisation of their properties, while also allowing artefacts of existing formalisms and tools to be cast in the theory.

We discuss our work on a contract theory, based in this principle. The contract theory is aimed at procedural, embedded software, and adheres to established desired properties in system design. It is defined only at the semantic level, in a denotational style. It is also parametrised on the semantic domain, enabling the instantiation of different semantics for particular uses. Previously, we have verified industrial embedded software deductively, with the goal of ensuring functional correctness of low-level system components. Recently we are also interested in high-level system properties, typically of a temporal nature. Because of the abstract view taken of contracts, both of these notions fit naturally into our framework.

3.15 Contracts in Static Verification

Rosemary Monahan (National University of Ireland – Maynooth, IE)

Joint work of Marie Farrell, Conor Reynolds, Rosemary Monahan
Main reference Marie Farrell, Conor Reynolds, Rosemary Monahan: "Using dafny to solve the VerifyThis 2021 challenges", in Proc. of the FTfJP 2021: 23rd ACM International Workshop on Formal Techniques for Java-like Programs, Virtual Event, Denmark, 13 July 2021, pp. 32–38, ACM, 2021.
URL https://doi.org/10.1145/3464971.3468422

We overview the structure of contracts languages used for specifying code for static verification. We report on our experience of using the Dafny Language in teaching software verification at both undergraduate and postgraduate level at Maynooth University; using Dafny to complete verification challenges in our research groups; and our experience of similar tools which have participated in the VerifyThis verification competition series over the past decade.

Contracts in static verification focus on modular specification of code. We present a range of examples and exercises with specification contracts consisting of pre-conditions, post-conditions and frame conditions, indicated by the `requires`, `ensures` and `modifies` clauses respectively. We discuss under-specification and over-specification in contracts, directing discussion to what is meant by contract verification. We note that verification proofs often require more information that that provided in the method contracts. This includes information expressed as assertions (`assert`), loop and class invariants (`invariant`), termination metrics (`decreases`) and specification-only `ghost` code, as well as using built-in data types (`set, sequence, multiset` etc.) which provide abstraction and support for verification. We discuss their role in both the specification contracts for the code, and in the contracts which verification tools use during the proof of correctness. Finally, we summarise how we view contract's in static verification and present future challenges for contract languages and static verification tools as the way that we build and verify software evolves.

References

1 M Farrell, C Reynolds, and R Monahan. *Using Dafny to Solve the VerifyThis 2021 Challenges* Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs, Association for Computing Machinery, 2021
2 M Farrell, M Bradbury, M Fisher, L Dennis, C Dixon, H Yuan and C Maple. *Using Threat Tnalysis Techniques to Guide Formal Verification: A Case Study of Cooperative Awareness Messages* International Conference on Software Engineering and Formal Methods, Springer, 2019
3 M Farrell, N Mavrakis, C Dixon and Y Gao. *Formal Verification of an Autonomous Grasping Algorithm* International Symposium on Artificial Intelligence, Robotics and Automation in Space, 2020
4 K Rustan M Leino. *Dafny: An automatic program verifier for functional correctness*. International Conference on Logic for Programming Artificial Intelligence and Reasoning, Springer, 2010
5 M Huisman, R Monahan, P Müller, A Paskevich, and G Ernst. *VerifyThis 2018: A Program Verification Competition*. Université Paris-Saclay Research Report, 2019
6 M Huisman, R Monahan, P Müller and E Poll. *VerifyThis 2016: A program verification competition* Enschede, the Netherlands: University of Twente, 2016
7 M Huisman, V Klebanov, R Monahan and M Tautschnig. *VerifyThis 2015* International Journal on Software Tools for Technology Transfer, Springer, 2017
8 M Huisman, R Monahan, P Müller, W Mostowski and M Ulbrich. *VerifyThis 2017: A Program Verification Competition* Karlsruhe Reports in Informatics, Karlsruhe Institute of Technology, 2017
9 M Huisman, V Klebanov and R Monahan. *VerifyThis 2012: A Program Verification Competition* International Journal on Software Tools for Technology Transfer, Springer, 2015

3.16 On Automatic Contract Synthesis

Philipp Rümmer (Uppsala University, SE)

Joint work of Anoud Alshnakat, Dilian Gurov, Christian Lidström, Philipp Rümmer
Main reference Anoud Alshnakat, Dilian Gurov, Christian Lidström, Philipp Rümmer: "Constraint-Based Contract Inference for Deductive Verification", pp. 149–176, Springer, 2020.
URL https://doi.org/10.1007/978-3-030-64354-6_6

This survey talk considers software contracts, consisting of pre-conditions and post-conditions, for imperative programs. Contracts of this kind enable modular verification of programs, and are therefore an important tool to scale up verification methods to large code bases. Formulating sufficient contracts is a time-consuming process, however, making the possibility to compute contracts automatically an attractive option. After studying the algebraic structure of software contracts, the talk surveys some of the contract inference methods that have been proposed in the literature. Considered contract inference methods derive contracts from programs implementations and program specifications, and can produce auxiliary annotations that complement manually written specifications.

The talk is partly based on recent joint work with Anoud Alshnakat, Jesper Amilon, Zafer Esen, Dilian Gurov, and Christian Lidström.

3.17 When You Hit the Frame

Thomas Santen (Formal Assurance – Aachen, DE)

Frame conditions of the form $x' = x$ are essential for pre/post condition contracts to specify the parts of a state that remain unchanged under execution of an operation. Though seemingly trivial equations, frame conditions are a common source of error in specifications of industrial systems, in particular when they need to be propagated through several levels of abstraction. The talk discusses common yet expensive error patterns and an approach to debug erroneous frame specifications.

3.18 Correctness-by-Construction: The CorC Ecosystem

Ina Schaefer (KIT – Karlsruher Institut für Technologie, DE)

Joint work of Ina Schaefer, Tabea Bordis, Tobias Runge, Alexander Kittelmann
Main reference Tabea Bordis, Loek Cleophas, Alexander Kittelmann, Tobias Runge, Ina Schaefer, Bruce W. Watson: "Re-CorC-ing KeY: Correct-by-Construction Software Development Based on KeY", in Proc. of the The Logic of Software. A Tasting Menu of Formal Methods – Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday, Lecture Notes in Computer Science, Vol. 13360, pp. 80–104, Springer, 2022.
URL https://doi.org/10.1007/978-3-031-08166-8_5

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

3.19 VeyMont: Parallelising Verified Programs instead of Verifying Parallel Programs

Petra van den Bos (University of Twente – Enschede, NL)

Joint work of Petra van den Bos, Sung-Shik Jongmans

We present VeyMont: a deductive verification tool that aims to make reasoning about functional correctness and deadlock freedom of parallel programs (relatively complex) as easy as that of sequential programs (relatively simple). The novelty of VeyMont is that

it "inverts the workflow": it supports a new method to parallelise verified programs, in contrast to existing methods to verify parallel programs. Inspired by methods for distributed systems, VeyMont targets coarse-grained parallelism among threads (i.e., whole-program parallelisation) instead of fine-grained parallelism among tasks (e.g., loop parallelisation).

4 Working groups

4.1 Two-state versus Trace-based Contracts: Report from Breakout Group 1

Dilian Gurov (KTH Royal Institute of Technology – Stockholm, SE), Bernhard Beckert (KIT – Karlsruher Institut für Technologie, DE), Alessandro Cimatti (Bruno Kessler Foundation – Trento, IT), Paula Herber (Universität Münster, DE), Srdan Krstic (ETH Zürich, CH), Martin Leucker (Universität Lübeck, DE), Christian Lidström (KTH Royal Institute of Technology – Stockholm, SE), Marco Scaletta (TU Darmstadt, DE)

Introduction

The original *design-by-contract* paradigm [6] has been proposed in the context of Hoare logic [4], which essentially relates initial with final states of a computation. While in Hoare logic, pre- and postconditions are usually used to put assertions on the initial and final state, Meyer used the broader concept of assumptions and guarantees. In this report, we refer to contracts that relate initial with final states as *two-state contracts*. In contrast, temporal properties are properties of sequences of states, called *traces*. Contracts about the temporal behaviour of components can therefore be termed *trace-based contracts*.

But when shall one use contracts of the first kind, and when of the second? Can they be combined? How can they be written, concretely, and with what formal semantics? How can they be formally verified, preferably automatically? These are some of the questions which Breakout Group 1 set out to explore.

Motivating Example

We discussed an example, where two interacting components, namely a *Sensor* and a *Controller* interact via an event, as shown with two simple finite state machines in Figure 1. In this example, a *Sensor* reads sensor values (action j_1) and performs some internal computations on them, e.g., filtering (j_2). Then, it sends a warning to the controller (*obstacle!* denotes sending event *obstacle*) if the read sensor value is below a given threshold ($x < 10$). Then, it goes to sleep for some time or until new sensor data is available. The controller runs concurrently with the sensor, and synchronizes on the event as a receiver (*obstacle*?). Whenever it receives an *obstacle* warning, it sets a local variable obstacle detected (*od*) to true, and performs some action i_2, for example to avoid the obstacle.

Classical contract-based approaches encapsulate the local effects of a component with a contract. A challenge of dynamically interacting components is that we want to reason about global properties based on the local ones. For example, whenever an obstacle is

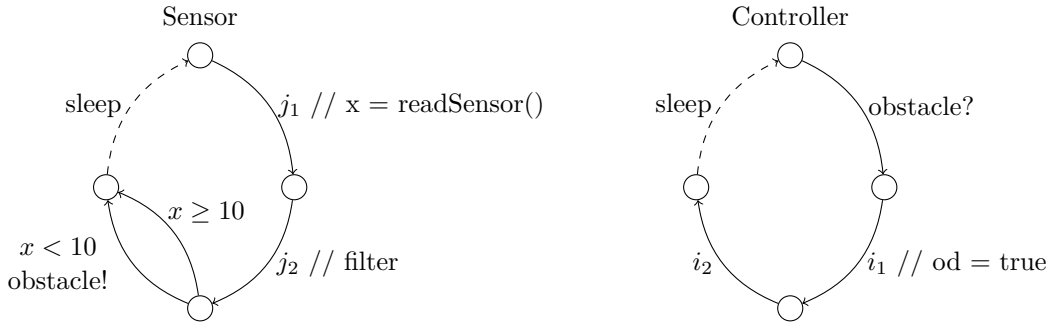

Figure 1 Interacting Concurrent Components.

detected ($x < 10$), there actually is a reaction of the controller, which can be expressed with the CTL formula $AG\,(x < 10 \rightarrow AF\,od)$. This global property involves variables from both the controller and the sensor. To encapsulate the local effects of the processes, possible abstractions for this example might merge internal computations into atomic code blocks (e.g., $c_i = i_1, i_2$ and $c_j = j_1, j_2$) and describe the behavior of the processes with behavioral specifications, e.g., as regular expressions $(c_j\, obstacle!)^*$ or $(obstacle?\, c_i)^*$, as recently proposed in [3].

Such a behavioral specification enables us to reason about properties that result from the local effects of both processes in combination, e.g., that the actions in c_j are actually followed by actions in c_i, because the processes are synchronized via the *obstacle* event. However, whether the property actually holds depends on the concrete synchronization mechanism, and the analysis is further complicated if a there runs a scheduler in the background, and we have asynchronous communication. Such behavioral specifications can be extended towards more complex properties; for example, a valve may open or close, take some time to open, might fail, might be scheduled and so forth, and intermediate steps might be relevant or not. We also discussed that it might be useful to have not just one contract, but a set of possible abstractions one can use for different purposes.

Two-state and Trace-based Contracts

To cope with complex, concurrent systems as shown above, a combination of two-state and trace-based contracts might be useful. We made the following observations and suggestions.

Contracts, Abstractly

At the most abstract level, a contract should specify the behaviour of a component along its *interface*. Conceptually, the *interface behaviour* of a component consists of the possible sequences of side-effects that invoking a component (e.g., procedure) gives rise to.

Two Aspects of Computation

There are conceptually two aspects of computation: *state transformation* and *interaction*. Two-state contracts are better suited, and have been designed, for the specification of the former, while trace-based contracts for the latter.

Specifying Contracts

Contracts are often stated as assumption-guarantee pairs. Assumptions can conceptually be of a temporal nature (such as, for example, in "a file should only be closed if it has been opened before that"), and could be specified in PLTL (past-time LTL), but could equivalently be expressed with a state formula, provided some "ghost" state is used to capture the "relevant part" of the history.

For specifying temporal contracts of procedural languages, logics of nested words may be useful (such as CaRet [1]), since they offer a construct that essentially amounts to the two-state contract corresponding to a procedure call.

Combining the Aspects

Complex systems typically exhibit both aspects of computation mentioned above. The aim of specification of such systems should be to abstract from the data transformation by *summarising* these with two-state contracts, and focus on the trace properties of the event sequences. We need methods and tools to help engineers to combine reasoning about data transformation with reasoning about event sequences (which, in separation, engineers know how to work with).

While trace-based properties are what we want to verify on a system level, two-state ones are rather on the local level and capture the concrete details of a component with finer granularity. Furthermore, the trace-based view should be used for interactions and interface specifications, and the two-state view for computations on data, i.e., state transformations. To connect these two views, contracts might be useful; for example, by means of two-state contracts one can *abstract* code blocks into TLA actions [5], and then apply model checking for the verification of temporal properties. This idea has recently been advocated in [2] by two of the present authors.

This idea could be embedded into a more general methodology for verification by building an abstract system view, as shown in Figure 2 for our motivating example. A preliminary approach to incorporate this idea into a verification process would be to perform three steps, considering the global and the local views separately:

1. Use interface specifications to get a trace-based system level view, while abstracting from the inner workings of each component; e.g., a behavioral specification of the overall system, as a composition of local views, could look for our example like this: $(c_j\ obs!)^*||(obs?\ c_i)^*$.
2. For each component: analyse locally, with a two-state view, under what assumptions a component adheres to its trace-based interface specification; e.g., as postcondition, our sensor has sent the obstacle warning under the condition $x < 10$ in the final state if, as precondition, it has initially been scheduled: $(scheduled, x < 10 \rightarrow obstacle!)$.
3. Verify the global property on the behavioral system specification, while using assumptions and guarantees from the local two-state based views as abstractions of the internal behavior of components, e.g. check that $AG\,(x < 10 \rightarrow AF\,od)$ is satisfied.

While the first step should focus on processes and their interactions, the second step should mainly capture local effects as a two-state view. With that, a separation of concerns and modular verification might be possible even for complex systems.

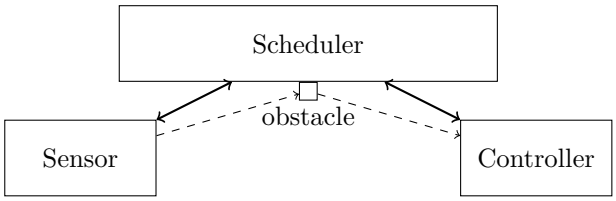

Figure 2 Abstract View with Contracts.

Conclusion

The discussion showed that contracts are used very differently in different communities. Depending on the interpretation of a contracts, and in particular, depending on whether a contract is interpreted as a two-state pair or a behavioral, trace-based specification, contracts play a very different role in program and system verification. Two-state contracts are very well-suited to capture local, transformational effects with no side-effects, while trace-based specifications are very well-suited to capture dynamic interactions and sequences of events.

We believe that *combining the two paradigms* has a high potential to modularize the verification effort in complex systems, and to increase the scalability and feasibility of formal methods. However, just throwing the two specification paradigms together might impede the feasibility of the underlying verification techniques, as the expressiveness of the contract language becomes too high. Instead, systematic abstractions and a separation of concerns are highly desirable.

References

1 Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004.

2 Jesper Amilon, Christian Lidström, and Dilian Gurov. Deductive verification based abstraction for software model checking. In *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022)*, volume 13701 of *Lecture Notes in Computer Science*, pages 7–28. Springer, 2022.

3 Gidon Ernst, Alexander Knapp, and Toby Murray. A hoare logic with regular behavioral specifications. In *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022)*, volume 13701 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2022.

4 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

5 Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

6 Bertrand Meyer. Applying "Design by contract". *IEEE Computer*, 25(10):40–51, 1992.

4.2 Abstraction in Contracts: Report from Breakout Group 2

Reiner Hähnle (TU Darmstadt, DE), Asmae Heydari Tabar (TU Darmstadt, DE), Eduard Kamburjan (University of Oslo, NO), Nikolai Kosmatov (Thales Research & Technology – Palaiseau, FR), Rosemary Monahan (National University of Ireland – Maynooth, IE), Thomas Santen (Formal Assurance – Aachen, DE), Ina Schaefer (KIT – Karlsruher Institut für Technologie, DE), Julien Signoles (CEA LIST – Gif-sur-Yvette, FR), Alexander J. Summers (University of British Columbia – Vancouver, CA)

Motivation

During the plenary discussions in the seminar it was universally agreed that the ability to specify data structures at a suitably abstract, implementation-independent level is crucial to keep specification and verification effort of complex programs manageable. At the same time, the abstraction mechanisms offered by contemporary contract languages are insufficient to achieve this goal.

In addition to feasibility of the specification and verification effort, the capability to abstract away from implementation details in specifications has further important advantages: (i) Better communication with stakeholders at their level of abstraction, (ii) encouragement to write abstract specifications first, (iii) independence from implementation language (multi-language support), (iv) independence from a specific implementation.

In consequence, the breakout group was tasked to explore what kind of abstraction mechanisms might be required and suitable.

Problem Description

Consider a piece of client code that intends to use the implementation of a given API. For example, the client needs to manage a *set of objects* of a certain type. Most languages offer an API for managing collections of objects, including sets. For efficiency reasons, sets are implemented with mutable data structures, such as arrays or linked lists. But this makes it problematic to specify the intended behavior at the *abstract* level, i.e. in terms of *sets*, because separation properties are specified at the implementation level and also differ among different implementations. In addition, one must make sure that the usage of the API by the client does not break memory separation.

In consequence, we need a discipline and mechanisms that encapsulate separation properties and that ensure their preservation at an abstract level. Current mechanisms as realized in languages such as Dafny or JML are based on *represents* clauses that define how abstract specification elements can be implemented in terms of concrete ones. While it is certainly essential to provide such representations, this is not sufficient to achieve abstraction in the sense outlined above for a number of reasons:

- The relation between abstract elements and the read/write footprint of concrete ones is not explicit, which is necessary to compute and verify assumptions.
- Represents clauses that relate to the same abstract datatype are not grouped together and named.
- There is no control over the visibility of the implementing elements.
- There are no specification constructs that permit to express under which conditions a given property of an abstract specification is preserved under a given operation.

$$\begin{array}{ccc} \mathrm{Valid}_{T_1^a}(V_1^a) \wedge \mathrm{Valid}_{T_2^a}(V_2^a) \wedge \mathrm{pre}^a(V_1^a, V_2^a) & \xrightarrow[\text{assigns } V_1^a, V_2^a]{f^a} & \mathrm{post}^a(\overline{V}_1^a, \overline{V}_2^a) \\ \Big\updownarrow R_{\mathrm{pre}} & & \Big\updownarrow R_{\mathrm{post}} \\ \mathrm{Valid}_{T_1^c}(V_1^c) \wedge \mathrm{Valid}_{T_2^c}(V_2^c) \wedge \mathrm{pre}^c(V_1^c, V_2^c) & \xrightarrow[\text{assigns } V_1^c, V_2^c]{f^c} & \mathrm{post}^c(\overline{V}_1^c, \overline{V}_2^c) \end{array}$$

Figure 3 Diagram for verification of concrete implementation with the help of abstract types.

Sketch of a Proposal

In consequence of the above considerations, we propose the concept of an *abstract type*.[1] This is not a type of the underlying implementation language and outside its type hierarchy. Rather, we are thinking of a trait-like structure that might look as follows:

```
abstract type T^a<P> {
  // 1. representation of type instance, invariants
  // 2. pure operations with representation
  // 3. impure operations with representation
  // 4. declare abstract specification building blocks:
  //      opaque, but extensible
}
```

This defines an abstract type T^a parameterized with some object type P (which we omit in the following for brevity). It provides

1. the representation of a T^a object in terms of concrete *implementing types* $T_1^c, \ldots, T_n^c$, together with invariants that define what a *valid* instance is, for example, the property that a set is idempotent. The predicate $\mathrm{Valid}_{T^a}(V^a)$ expresses that V^a is a valid value of T^a.
2. a set of pure operations that do not change the state and which can be used in assertions.
3. a set of impure operations.
4. a subset of pure and impure operations to be used as building blocks in abstract specifications. The remaining operations and the definitions are opaque.

Let us call the relation between abstract and concrete types as sketched above a *coupling*. We will also need to define *abstract properties* that relate one or more abstract object:

```
abstract property prop(T_1^a,...,T_n^a) {
  // representation in terms of concrete properties;
  // invariants
}
```

Abstract properties are domain-specific and should be defined in connection with a given abstract type. Examples of properties are: separation, reachability, order, containment, etc.

We illustrate how this setup can be used to simplify verification of concrete types, i.e. the *implementation* of an abstract type. Consider the bottom row in Fig. 3, representing the following contract of a concrete implementation of function f^c:

[1] Their might be a better terminology, this is just a working proposal.

```
f^c {
    requires    Valid_{T_1^c}(V_1^c) ∧ Valid_{T_2^c}(V_2^c) ∧ pre^c(V_1^c, V_2^c);
    ensures     post^c(V̄_1^c, V̄_2^c);
    assignable {V_1^c, V_2^c};
}
```

Assume we want to prove this contract without referring to the concrete implementation as much as possible. We can assume that the corresponding *abstract* contract for f^a (analogous to f^c) has been shown already. Moreover, we can assume the coupling relation

$$R_{\text{pre}} \equiv \bigwedge_{i=1,2} \big(\text{Valid}_{T_i^a}(V_i^a) \wedge \text{Valid}_{T_i^c}(V_i^c) \wedge [\text{coupling of } V_i^a,\ V_i^c \text{ via } T_i^a]\big)$$

to be given. The corresponding relation R_{post} can be computed using the implementation of f^c.

Now we establish the preservation property for pre and post:

$$R_{\text{pre}} \longrightarrow \big(\text{pre}^a(V_1^a, V_2^a) \longleftrightarrow \text{pre}^c(V_1^c, V_2^c)\big) \tag{1}$$

$$R_{\text{post}} \longrightarrow \big(\text{post}^a(\overline{V}_1^a, \overline{V}_2^a) \longleftrightarrow \text{post}^c(\overline{V}_1^c, \overline{V}_2^c)\big) \tag{2}$$

If the properties are chosen well and with the help of some composition theorems, this needs to be shown only once.

This should be sufficient to establish the concrete contract of f^c, i.e. the following should be provable as a meta theorem:

$$R_{\text{pre}} \wedge R_{\text{post}} \wedge \text{pre}^c(V_1^c, V_2^c) \longrightarrow \text{post}^c(\overline{V}_1^c, \overline{V}_2^c)\ .$$

Conclusion and Open Questions

Obviously, this is just a sketch, not a formal proof. To go further, we need to instantiate the schema for specific frameworks and look at a concrete example.[2] Notably, it must be clarified in which way our proposal generalizes model fields and methods. The meta theorem(s) must be formally proven, the assumptions clarified, etc.

Besides this, there are several open questions:

- What are minimal assumptions to make this work? Can the equivalence in eqs. (1), (2) be relaxed?
- What are the links to the B-method, as well to abstract interpretation?
- We assumed that the footprints of the V_i^c are disjoint – can it be generalized?
- To be practicable, the framework sketched here should be supported by specification idioms and patterns.

[2] During the breakout sessions, Alexander Summers sketched how a similar setup might look in the VIPER framework.

4.3 Specification Engineering: Report from Breakout Group 3

Wolfgang Ahrendt (Chalmers University of Technology – Göteborg, SE), David Cok (Safer Software Consulting – Rochester, US), Gidon Ernst (LMU München, DE), Sophie Lathouwers (University of Twente – Enschede, NL), Giles Reger (University of Manchester, GB), Kristin Yvonne Rozier (Iowa State University – Ames, US), Philipp Rümmer (Uppsala University, SE)

We document the outcome of the break-out sessions on Specification Engineering, held at the Dagstuhl Seminar on Principles of Contract Languages, November 2022.

Research Question

We identify two main activities in engineering the (formal) specification of a system component or unit: a) the coming about of some first version of a specification, and b) developing existing specifications further, by maintenance, correction, refinement, extension, and alike. Activity a) refers to the initial formalization of the component's requirements, which could be based on natural language descriptions, on the specifiers own understanding of the requirements, or other sources. While this is a very relevant problem, it has not been the focus of our discussions, and is not what this summary contributes to. Instead, we focus entirely on b), i.e., the activity on assessing the quality of a given status of the specification, and developing it further accordingly.

Specifications ought to represent certain desired properties of the system, typically as constraints, postulates, and models of the behavior. Those should be *more obviously correct than the code itself*, and should *sufficiently describe the guarantees that other components rely on.*

Large parts of the realm of formal methods is devoted to providing formal evidence that a system or component actually implements its specification, through mechanized verification. As of yet, much less focus is given to providing evidence on the correspondence between intended requirements and the formal specification, even if there is high demand on addressing this issue. Accordingly, we are interested in *the correspondence between the mental model inside the head of developers and the model that is actually formalized as by the specification.* Moreover, we are interested in *the correspondence between the what properties other components require, and what the specification of this component actually guarantees.*

While there are some established techniques, e.g. from the area requirements validation such as illustrating the meaning of temporal logic formulas, we seek to integrate these into a general framework that supports a systematic tool-driven process to ensure the aforementioned correspondences. Therefore, our research goal is

- to bridge between the mental model of the system's *intended* behavior and the *formalized* specification, and
- to bridge between requirements *assumed* by other components, and the *guarantees* provided by this specification.

We have identified the following main questions.

- How is the process driven? E.g., which (kinds of) questions should the developer be able to pose to the system's specification to get a better understanding, and similarly, which questions should be derived from the specification to be posed to the developer?
- What are the different dimensions and artifacts over which this dialogue is formulated and how can we ensure that these have a formally precise meaning while at the same time being easy to understand?
- What tools exists today that can be leveraged at which stages of the process?
- What tools (or tool functionalities) are yet to be provided by the community in the course of realizing such a process?

A method for specifications evolution

To address the bridge between the mental model of the system's *intended* behavior and the *formalized* specification, we propose to use an iterative approach similar to active learning, see Figure 4.

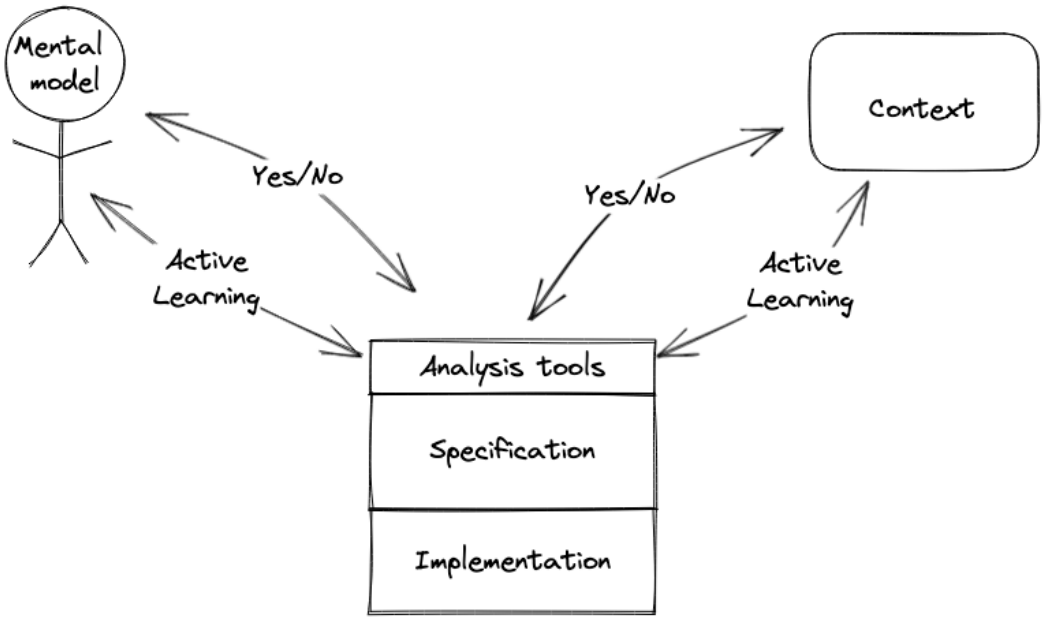

Figure 4 Active learning approach to refine specifications where different tools and techniques are used to iteratively improve the existing specification.

The main idea behind this approach is that the user can iteratively improve upon the existing specification by using different tools and techniques to gain a better understanding of the current specification, and develop it further accordingly. Similarly, the context in which the system is used can use a similar approach to enforce limitations of the system's context. Focusing on the relation of the user's mental model and the specification for now, the approach caters for tool supported querying in two directions. In the one direction, the user can post queries, to be answered by tools analyzing the specification. These answers shall help the user to evaluate how well the specification matches her own mental model of the component. In the other direction, tools can generate queries to the user, for instance by deducing consequences of the specification. The user can "answer" the query by indicating

whether the that consequence matches the intention. Also, tools can question the use of a common logical fallacies, anti-patterns, inconsistencies or vacuously true parts of the specification.

When asking questions to the user in this setting, it is important to take a few things into account. Firstly, one should ask the most important questions first. To achieve this one could use dynamic question prioritization. Secondly, one should be efficient with the questions that are asked to avoid overburdening the user. Finally, one should ask both positive and negative questions.

Techniques and tools for specification analysis

Some examples of techniques that can be used in this active learning setting to gain a better understanding of the specification are exemplified below. These techniques are well-known and their integration into our method will benefit from advances in the respective research areas.

Generating models of specifications helps to make concrete what specifications mean. The existence of models in the first place rules out vacuous specifications, i.e., formalizations that admit no behavior at all (e.g. an unsatisfiable precondition renders a method contract vacuously true) or that trivially allow all behaviors. Such models have a wide variety of uses cases, cf. below, and can be generated with the help of modern SAT/SMT solvers.

Visualization is a powerful tool that helps to illustrate the meaning of specifications. This has notably been explored for temporal logic formulas, where e.g. input signals for cyber-physical systems can be generated automatically and presented side-by-side with the corresponding outputs. Such a visual presentation has the advantage that it is very easy for a human to assess whether it matches their mental model.

The original formulation of the specification as provided by the engineer can be transformed symbolically and *re-phrased* by automated tools. Examples are the simplification and normalization of formulas and the computation of logical consequences and lemmas via theory exploration. This help to clarify what the specification means in the general sense, in contrast to examples which can only cover concrete cases.

Linter-like checks can be used to discover typical formalization mistakes. For instance, a universal quantifier should always be paired with an implication and an existential quantifier should be paired with a conjunction, as an example, the formula $\exists x.\ x \in P \implies Q(x)$ almost certainly does not express the engineer's intention. Such checks can be integrated with a database of typical specification patterns and anti-patterns. Another example are typical liveness and safety properties in temporal logic.

Mutation testing is technique that attempts to discover the effect of changing the specification in certain "typical" ways (e.g. to replace an occurrence of $<$ by $\leq$). In software testing this is applied to assess the robustness of test-suites, here it can similarly be used to discover whether such mutations are in conflict with some insights learned already as evidence that the correspondence between the mental model and the specification has been exercised in sufficient detail to notice the mutation at all.

This approach can be used for many different types of specifications, e.g. pre-/postcondition style contracts, temporal logic formulas, or test cases. Depending on the type of specification that you are interested in, different techniques will be available in this iterative approach.

The Way Forward

In order to move forward in the direction of realizing the method sketched above, one should survey existing tools for analyzing specifications and answering queries about them. Equally important would be an investigation on what kind of queries (and their according answers) would help users to judge the extent to which a current specification matches the mental model.

A proof of concept for the proposed method could be to instantiate the approach with at least two concrete ways, e.g., with tow tools supporting to query pre/post-specifications or trace-based specifications, respectively. Those shall then be applied to a few small case studies to begin with. This will help to demonstrate how current methods and tools can already support such a process and where gaps and weaknesses are.

4.4 Interoperable Contracts: Report from Breakout Group 4

Petra van den Bos (University of Twente – Enschede, NL), Claire Dross (AdaCore – Paris, FR), Sylvain Hallé (University of Quebec at Chicoutimi, CA), Marieke Huisman (University of Twente – Enschede, NL), Marie-Christine Jakobs (TU Darmstadt, DE), Mattias Ulbrich (KIT – Karlsruher Institut für Technologie, DE)

This document summarises the discussions of the break out session on Interoperable Contracts held at the Dagstuhl Seminar on Principles of Contract Languages.

Research Question

The current status in research on contracts is that there is a versatile set of tools for verifying software, but these tools cannot be used interchangeably. We explore what would be needed to achieve interoperability. Specifically, our research question is:

What contract infrastructure is needed to allow tools to interoperate w.r.t. formal contracts?

We consider this question for a wide range of tools. Besides tools using pre- and postcondition contracts, we also include tools with other notions of contracts for software. A motivation for including these tools is that they also target verification of software. In Table 1 we list the research areas of the type of tools we consider. For each area, we state the most important 'types' of contract, i.e. what needs to be specified to enable the software verification. Additionally, the table lists the mode of operation: is software analysed statically, or at run-time, so dynamically.

From the overview of Table 1 we selected two specific interoperability cases:

Pre-postcondition contracts only Interoperability for two deductive verification tools, or a deductive verification and a run-time assertion checking tool, using pre- and postcondition contracts as specification language.

Different contract types + static-dynamic Interoperability between a deductive verification tool and a dynamic verification tool using automata and/or temporal logic specifications.

Table 1 Research areas of software verification tools.

Research area	Included contract types	Mode
Deductive verification	Pre- and postconditions, invariants	Static
Software model checking	Reachable code points, pre- and post-conditions, assertions	Static
Classical model checking	Automata, temporal logic	Static
Run-time assertion checking	Pre- and postconditions, assertions	Dynamic
Run-time verification	Automata, temporal logic, regexes	Dynamic
Automated testing	Automata	Dynamic

Interoperability between Tools using Pre-Postcondition Contracts

Next, we enumerate the main challenges that we identified for interoperability between deductive verification tools and run-time assertion checkers, using pre- and postcondition contracts as specification language.

Contract language and translation. Tools use languages with different syntax and semantics. Translation from one tool language to the other is needed to make tools interoperable. A promising direction is to design a common or unified language for contract exchange, though this could be "too late now" with the current diverse landscape.

Executable static languages. There are tools with languages that can be used for both static and dynamic verification of pre-postcondition contracts, e.g. the E-ACSL language component of ACSL in tool Frama-C, and the language of the tool Spark have been designed such that all specifications can be used for static verification, but are also executable for run-time verification. However, this requires that the static and dynamic semantics of the assertions coincide, which makes it challenging.

Implicit assumptions. When exchanging verification tasks, implicit assumptions need to be taken into account, as an assumption can change the semantics of a contract. We distinguish three types of differences in assumptions:

- Differences in semantics that do not cause any issue for the combination. For example, one tool does not verify exceptional behaviour like run-time exceptions, so it must be ensured that this part is verified by another tool that is able to do this.
- Differences that need to be made explicit in the pre/postcondition, but still allow the interaction to take place. For example, if one tool assumes that objects are non-null, then an explicit assertion can be used in another tool without this assumption.
- Different constructs that must be disallowed because they are handled in a different way by the two tools. For example, different semantics for floating point numbers are used in two different tools, so the verification result is different for the two tools.

Division of the verification task. To be able to use the interoperability of tools, we need a method for dividing the verification task. Here, the contract, the code, or both could be divided in pieces for distribution to different tools.

Coordinating tool interoperability. It needs to be decided, either manually or automatically, which tool verifies what. Note here that to be able to use different tools effectively, an explicit overview or mapping of tool capabilities is needed, to know which parts or properties need to be verified by which tool for obtaining the most optimal (e.g. most conclusive or most time-efficient) verification result.

Combining verification results. The verification results of the used tools need to be combined to one global verification result saying what was (not) proven. Used assumptions need to be included in this result. Also, in case of verification failures, it must be possible to to trace back the failures for analysis and modification purposes.

Application to multi-language software. A software system may consist of multiple languages. Verifying this with a single verification tool is already challenging.

Static-Dynamic Interoperability for Different Contract Types

Below we list approaches for interoperability between a tool using contracts, and a tool using automata and/or temporal logic. We note that most approaches can be found in the existing literature and that static-dynamic interoperability may be coordinated in both directions.

Outsourcing. A deductive verifier could outsource difficult-to-prove conditions to a (runtime) monitor or assertion checker, or a runtime monitor could outsource difficult-to-monitor conditions to a deductive verifier.

Monitor reduction. Deductive verification or model checking can be used to check properties such that monitoring assertions can be removed. For example, one might omit monitoring a part of a temporal logic formula because it has been proven via static verification.

Contracts for abstraction hierarchy. One may replace a subcomponent of an automaton by a pre- post condition contract or a temporal logic formula and use a model checker to verify this abstraction step.

Contracts for automaton actions. One may use an automaton to model the sequences of method calls, where a method corresponds to a transition, specified by a contract: the precondition is the guard of the transition, and the postcondition the action or resulting state change.

Correspondence checks between automaton and code via annotations. One may translate an automaton to annotations in the program to prove correspondence between the automaton and the code.

Verification of unbounded data in testing. One may use deductive verification to abstract from unbounded data parameters in an automaton used for test generation.

Follow-Up Plans

Several of the participants in the working group would like to explore the interoperability of tools that use pre- and postconditions as contracts more. They plan to discuss more about these different categories of implicit assumptions, and whether these could be identified in a systematic manner.

For the combination of static and dynamic verification techniques with different contract types, the results were more inconclusive. It is an interesting area, with a high potential, but we need further discussions to come up with concrete new ideas on how to exploit this combination.

Participants

- Wolfgang Ahrendt
Chalmers University of Technology – Göteborg, SE
- Bernhard Beckert
KIT – Karlsruher Institut für Technologie, DE
- Alessandro Cimatti
Bruno Kessler Foundation – Trento, IT
- David Cok
Safer Software Consulting – Rochester, US
- Claire Dross
AdaCore – Paris, FR
- Gidon Ernst
LMU München, DE
- Dilian Gurov
KTH Royal Institute of Technology – Stockholm, SE
- Reiner Hähnle
TU Darmstadt, DE
- Sylvain Hallé
University of Quebec at Chicoutimi, CA
- Paula Herber
Universität Münster, DE
- Asmae Heydari Tabar
TU Darmstadt, DE
- Marieke Huisman
University of Twente – Enschede, NL
- Marie-Christine Jakobs
TU Darmstadt, DE
- Eduard Kamburjan
University of Oslo, NO
- Nikolai Kosmatov
Thales Research & Technology – Palaiseau, FR
- Srdan Krstic
ETH Zürich, CH
- Sophie Lathouwers
University of Twente – Enschede, NL
- Martin Leucker
Universität Lübeck, DE
- Christian Lidström
KTH Royal Institute of Technology – Stockholm, SE
- Rosemary Monahan
National University of Ireland – Maynooth, IE
- Doron A. Peled
Bar-Ilan University – Ramat Gan, IL
- Giles Reger
University of Manchester, GB
- Kristin Yvonne Rozier
Iowa State University – Ames, US
- Philipp Rümmer
Uppsala University, SE
- Thomas Santen
Formal Assurance – Aachen, DE
- Marco Scaletta
TU Darmstadt, DE
- Ina Schaefer
KIT – Karlsruher Institut für Technologie, DE
- Julien Signoles
CEA LIST – Gif-sur-Yvette, FR
- Alexander J. Summers
University of British Columbia – Vancouver, CA
- Mattias Ulbrich
KIT – Karlsruher Institut für Technologie, DE
- Petra van den Bos
University of Twente – Enschede, NL