



# A Maneuver-Centric Formal Engineering Approach for Cyber-Physical Systems

Von der  
Carl-Friedrich-Gauß-Fakultät  
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines  
Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von  
Alexander Kittelmann  
geboren am 14. Juli 1988  
in Osnabrück

Eingereicht am: 14.12.2021  
Disputation am: 25.02.2022  
1. Referentin: Prof. Dr.-Ing. Ina Schaefer  
2. Referent: Prof. Dr. rer. nat. Reiner Hähnle  
3. Referent: Prof. Dr. Einar Broch Johnsen



# Abstract

In present days, cyber-physical systems (CPSs) play an increasingly vital role in our everyday lives and effectively demonstrate how we, as a society, aim at leveraging technology to mitigate human error. Automated vehicles, robots, and aircraft are only a few prime examples of such systems that combine digital computations, such as control algorithms, with physical motion in space. As such systems become more complex, designing them becomes more challenging. The main reason is their inherently safety-critical and hybrid nature, which not only requires highly standardized quality assurance of the running software, but also sophisticated preplanning. However, these two challenges come with a cost, such as scalability issues during the software development process or the need for high expertise. Consequently, we hypothesize that the key to designing safe behaviors for CPSs in a manageable fashion and maximizing trust early on mandates a software engineering process that (1) prioritizes appropriate abstractions in the early design phase (i.e., starting with requirements engineering), but (2) builds formal safety proofs from smaller parts at each stage of the process to establish correctness of the whole system. In this thesis, we make several contributions to establish such a formal engineering process, where safety proofs at each development step are built by applying deductive verification. In particular, we focus on three key areas namely (1) planning, modeling, and analyzing safe behavior of cyber-physical systems, (2) deriving correct implementations with the goal of validating them through simulation, and (3) improving deductive verification applied on source-code level.

As a starting point for modeling safe CPS behavior, we introduce a formal framework based on *skill graphs*, a graph-based modeling formalism for decomposing the overall behavior of CPSs into isolated and verifiable *maneuvers* (e.g., following a leading vehicle in the context of automated driving). For specifying and verifying skill graphs, we combine them with a contract theory and model parts of their behavior by means of *hybrid programs*. Crucially, skill graphs can be composed to exhibit more complex maneuvers built from simpler ones. This *compositional* nature of our framework enables the reduction of verification complexity while it simultaneously prioritizes proof reuse.

Furthermore, we introduce a methodology for refining verified skill graphs to component-based architectures. In particular, nodes in a skill graph are mapped to components and edges correspond to component connectors. While certain components can be generated automatically, safe component behavior of to-be-developed components is ensured by applying the *correctness-by-construction* paradigm. Subsequent simulations of the executable and verified maneuvers allow to validate their requirements in a diverse set of scenarios.

Finally, we discuss two techniques in the context of formal specification and deductive program verification with the goal of bridging the gap between software engineering practices and formal methods. First, we assess to what extent a mutation analysis is valid and practical for measuring the precision of software contracts, which is an invaluable metric for software developers working with formal specifications. Second, we present a technique to increase the performance and success rate of the automatic proof search of *configurable* program verifiers by examining the influence of control parameters.



# Zusammenfassung

Cyber-physische Systeme nehmen eine zunehmend wichtige Rolle in unserem Alltag und unserer Gesellschaft ein, und haben das Ziel, durch technologischen Fortschritt und Automatisierung menschliches Fehlverhalten zu minimieren. Zu den Einsatzbereichen gehören autonome Straßenfahrzeuge, autonome Roboter, oder aber auch Luftfahrzeuge. Cyber-physische Systeme vereinen dabei digitale Software mit mechanischen und elektrischen Komponenten im physischen Raum, was zu schwerwiegendem Fehlverhalten in der realen Welt führen kann. Durch die steigende Komplexität erhöht sich ebenfalls auch das benötigte Expertenwissen während der Entwicklungsphase solcher Systeme.

Um dennoch die Sicherheit zu gewährleisten, muss besonderes Augenmerk auf die Qualitätssicherung im Entwicklungsprozess gelegt werden. Vorzugsweise werden hier Konzepte der formalen Verifikation angewendet, die eine Zertifizierung der Korrektheit der ausgelieferten Software garantieren. Der inhärente Nachteil ist allerdings die schlechte Skalierung und das benötigte Expertenwissen solcher Methoden. Wir nehmen daher an, dass ein erfolgreicher Entwicklungsprozess mindestens zwei Konzepte miteinander vereint: hohe Abstraktion in der frühen Entwicklungsphase und formale, modulare Beweisführung kleinerer Komponenten mit dem Ziel die Korrektheit des Gesamtsystems zu attestieren. Diese Arbeit leistet wichtige Beiträge, um sich einen solchen formalen Entwicklungsprozess anzunähern. Insbesondere werden dabei drei Kernbereiche beleuchtet: (1) Planung, Modellierung, und beweisgestützte Verifikation von Cyber-physischen Systemen während der Designphase, (2) Ableitung korrekter Implementierungen, die virtuell simuliert werden können, und (3) Verbesserung der Anwendung der deduktiven Verifikation für End-Nutzer auf Implementierungsebene.

Um sicheres Verhalten modellieren und verifizieren zu können, entwickeln wir in dieser Arbeit zunächst ein formales Konzept basierend auf *Fähigkeitsgraphen* (engl.: skill graphs). Fähigkeitsgraphen stellen eine grafische Notation zur Beschreibung von isolierten Fahrmanövern dar (z.B. Abstandsautomat), die dabei besonderen Fokus auf Modularität legt. Zur Spezifikation und Verifikation kombinieren wir Fähigkeitsgraphen mit *Verträgen* und *hybriden Programmen*. Die Modularität von Fähigkeitsgraphen erlaubt es unter bestimmten Bedingungen, komplexere Manöver aus simpleren abzuleiten und Verifikationsresultate zu transferieren. Dies reduziert nicht nur die Beweiskomplexität, sondern verbessert auch die Wiederverwendung von bereits geleistetem Beweisaufwand.

Neben der Verifikation solcher Modelle ist es auch wichtig, die grundlegenden Sicherheitsanforderungen auf Vollständigkeit hin zu validieren, was durch Simulation (d.h., virtuelle Ausführung unter Realbedingungen) der Manöver erreicht werden soll. Dafür stellen wir eine Methodik vor, um aus Fähigkeitsgraphen komponentenbasierte Architekturen abzuleiten. Während ein Großteil dieser Ableitung automatisiert werden kann, so gibt es durch die hohe Abstraktion der Fähigkeitsgraphen auch manuell fertig zu entwickelnde Komponenten. Um dennoch Korrektheit garantieren zu können, kombinieren wir diese Methodik mit dem *Correctness-by-Construction* Paradigma.

Schlussendlich diskutieren wir zwei Probleme im Kontext der vertragsbasierten Spezifikation und deduktiven Verifikation mit dem Ziel, die vorherrschende Lücke zwischen Praktiken der Soft-

warenentwicklung und den Formalen Methoden zu verkleinern. Als erstes untersuchen wir, inwiefern eine *Mutationsanalyse* Erkenntnisse über die *Vollständigkeit* vertragsbasierter Spezifikationen ermöglicht. Anschließend präsentieren wir eine Technik, welche zum Ziel hat, die Effektivität und Effizienz des automatischen Beweisalgorithmus' konfigurierbarer Programmbeweiser zu steigern, indem wir den Einfluss von Kontrollparametern untersuchen.

# Acknowledgements

In the past five years, a large number of people have supported me in this exciting journey. The following words can only scratch the surface of how much your support really means to me, but know that I am truly thankful for your time, guidance, wise words, jokes, and love.

First of all, I deeply thank my supervisor, Ina Schaefer, for giving me the chance to pursue a Ph.D. in an excellent research and learning environment, and for being a true mentor both academically and personally. I am also deeply grateful towards my previous colleague Thomas Thüm, now a professor himself, who supervised my master's thesis and eventually brought me on board. His mentoring in the first three years allowed me to find my own voice in academia. I thank my reviewers, Prof. Reiner Hähnle and Prof. Einar Broch Johnson, for taking the time to review my dissertation and providing me with valuable feedback.

I am especially honored to have worked with such nice and enthusiastic colleagues at the work-group *Software Engineering and Automotive Informatics* at TU Braunschweig, most of whom I now consider as friends. I will never forget the rounds of table football that were sometimes even at the verge of ending friendships (just kidding...), and the phenomenal christmas parties. These are some of my most valuable memories. Special thanks to Christoph Seidl, who always took time to help me with my career, and the *formal methods club* members Tobias Runge and Tabea Bordis, with whom I worked together the closest.

Finally, I thank my family for their tremendous support: my mother, Martina, who gave her all to raise me in a single-parent household before finding new company, and my beloved wife, Diana, who supported (and beared with) me during the research phase and writing process. An honorable mention goes to our puppies, Emily and Fuchur, which are the best after all. While Emily was with us since 2016, Fuchur joined us early pandemic. He taught me the essence of *live and let live* and to enjoy life amid such adverse circumstances. I will be forever thankful for that.





# Publications

This doctoral thesis is mainly based on the following peer-reviewed publications. Please note that I authored all mentioned publications under my former family name *Knüppel*.

- [1] **Knüppel, A.**, L. Schaer, and I. Schaefer (2021a). “How much Specification is Enough? Mutation Analysis for Software Contracts”. In: *Proc. in the Intl. Conference on Formal Methods in Software Engineering (FormalISE)*. Ed. by S. Bliudze, S. Gnesi, N. Plat, and L. Semini. IEEE, pp. 42–53. DOI: [10.1109/FormalISE52586.2021.00011](https://doi.org/10.1109/FormalISE52586.2021.00011). URL: <https://doi.org/10.1109/FormalISE52586.2021.00011>.
- [2] **Knüppel, A.**, T. Thüm, and I. Schaefer (2021b). “GUIDO: Automated Guidance for the Configuration of Deductive Program Verifiers”. In: *Proc. in the Intl. Conference on Formal Methods in Software Engineering (FormalISE)*. Ed. by S. Bliudze, S. Gnesi, N. Plat, and L. Semini. IEEE, pp. 124–129. DOI: [10.1109/FormalISE52586.2021.00018](https://doi.org/10.1109/FormalISE52586.2021.00018). URL: <https://doi.org/10.1109/FormalISE52586.2021.00018>.
- [3] **Knüppel, A.**, I. Jatzkowski, M. Nolte, T. Thüm, T. Runge, and I. Schaefer (2020a). “Skill-Based Verification of Cyber-Physical Systems”. In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Ed. by H. Wehrheim and J. Cabot. Vol. 12076. Lecture Notes in Computer Science. Springer, pp. 203–223. DOI: [10.1007/978-3-030-45234-6\\_10](https://doi.org/10.1007/978-3-030-45234-6_10). URL: [https://doi.org/10.1007/978-3-030-45234-6\\_10](https://doi.org/10.1007/978-3-030-45234-6_10).
- [4] **Knüppel, A.**, T. Runge, and I. Schaefer (2020b). “Scaling Correctness-by-Construction”. In: *Proc. of the Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Ed. by T. Margaria and B. Steffen. Vol. 12476. Lecture Notes in Computer Science. Springer, pp. 187–207. DOI: [10.1007/978-3-030-61362-4\\_10](https://doi.org/10.1007/978-3-030-61362-4_10). URL: [https://doi.org/10.1007/978-3-030-61362-4\\_10](https://doi.org/10.1007/978-3-030-61362-4_10).
- [5] **Knüppel, A.**, T. Thüm, C. I. Pardylla, and I. Schaefer (2018a). “Understanding Parameters of Deductive Verification: An Empirical Investigation of KeY”. In: *Proc. of the Intl. Conference on Interactive Theorem Proving (ITP)*. Ed. by J. Avigad and A. Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Springer, pp. 342–361. DOI: [10.1007/978-3-319-94821-8\\_20](https://doi.org/10.1007/978-3-319-94821-8_20). URL: [https://doi.org/10.1007/978-3-319-94821-8\\_20](https://doi.org/10.1007/978-3-319-94821-8_20).
- [6] **Knüppel, A.**, T. Thüm, C. Pardylla, and I. Schaefer (2018b). “Experience Report on Formally Verifying Parts of OpenJDK’s API with KeY”. In: *Proc. of the Intl. Workshop on Formal Integrated Development Environment (F-IDE)*. Ed. by P. Masci, R. Monahan, and V. Prevosto. Vol. 284. EPTCS, pp. 53–70. DOI: [10.4204/EPTCS.284.5](https://doi.org/10.4204/EPTCS.284.5). URL: <https://doi.org/10.4204/EPTCS.284.5>.

### Further peer-reviewed publications directly and indirectly related to this thesis.

- [7] Bordis, T., T. Runge, **A. Knüppel**, T. Thüm, and I. Schaefer (2020). “Variational correctness-by-construction”. In: *Proc. of the International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS)*. Ed. by M. Cordy, M. Acher, D. Beuche, and G. Saake. ACM, 7:1–7:9. DOI: [10 . 1145 / 3377024 . 3377038](https://doi.org/10.1145/3377024.3377038). URL: <https://doi.org/10.1145/3377024.3377038>.
- [8] Schaefer, I., T. Runge, **A. Knüppel**, L. Cleophas, D. G. Kourie, and B. W. Watson (2018). “Towards Confidentiality-by-Construction”. In: *Proc. of the Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Ed. by T. Margaria and B. Steffen. Vol. 11244. Lecture Notes in Computer Science. Springer, pp. 502–515. DOI: [10 . 1007 / 978 - 3 - 030 - 03418 - 4 \\\_ 30](https://doi.org/10.1007/978-3-030-03418-4_30). URL: [https://doi.org/10.1007/978-3-030-03418-4\\_30](https://doi.org/10.1007/978-3-030-03418-4_30).
- [9] Runge, T., **A. Knüppel**, T. Thüm, and I. Schaefer (2020). “Lattice-Based Information Flow Control-by-Construction for Security-by-Design”. In: *Proc. in the Intl. Conference on Formal Methods in Software Engineering (FormalISE)*. ACM, pp. 44–54. DOI: [10 . 1145 / 3372020 . 3391565](https://doi.org/10.1145/3372020.3391565). URL: <https://doi.org/10.1145/3372020.3391565>.
- [10] Thüm, T., **A. Knüppel**, S. Krüger, S. Bolle, and I. Schaefer (2019). “Feature-oriented contract composition”. *J. Syst. Softw.* 152, pp. 83–107. DOI: [10 . 1016 / j . jss . 2019 . 01 . 044](https://doi.org/10.1016/j.jss.2019.01.044). URL: <https://doi.org/10.1016/j.jss.2019.01.044>.
- [11] **Knüppel, A.**, S. Krüger, T. Thüm, R. Bubel, S. Krieter, E. Bodden, and I. Schaefer (2020c). “Using Abstract Contracts for Verifying Evolving Features and Their Interactions”. In: *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*. Ed. by W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, and M. Ulbrich. Vol. 12345. Lecture Notes in Computer Science. Springer, pp. 122–148. DOI: [10 . 1007 / 978 - 3 - 030 - 64354 - 6 \\\_ 5](https://doi.org/10.1007/978-3-030-64354-6_5). URL: [https://doi.org/10.1007/978-3-030-64354-6\\_5](https://doi.org/10.1007/978-3-030-64354-6_5).
- [12] **Knüppel, A.**, T. Thüm, C. Pardylla, and I. Schaefer (2018c). “Scalability of Deductive Verification Depends on Method Call Treatment”. In: *Proc. of the Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Ed. by T. Margaria and B. Steffen. Vol. 11247. Lecture Notes in Computer Science. Springer, pp. 159–175. DOI: [10 . 1007 / 978 - 3 - 030 - 03427 - 6 \\\_ 15](https://doi.org/10.1007/978-3-030-03427-6_15). URL: [https://doi.org/10.1007/978-3-030-03427-6\\_15](https://doi.org/10.1007/978-3-030-03427-6_15).
- [13] **Knüppel, A.**, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer (2017). “Is there a mismatch between real-world feature models and product-line research?” In: *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*. Ed. by E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman. ACM, pp. 291–302. DOI: [10 . 1145 / 3106237 . 3106252](https://doi.org/10.1145/3106237.3106252). URL: <https://doi.org/10.1145/3106237.3106252>.
- [14] Schlie, A., **A. Knüppel**, C. Seidl, and I. Schaefer (2020). “Incremental feature model synthesis for clone-and-own software systems in MATLAB/Simulink”. In: *Proc. of the Intl. Systems and*

*Software Product Line Conference (SPLC)*. Ed. by R. E. Lopez-Herrejon. ACM, 7:1–7:12. DOI: [10.1145/3382025.3414973](https://doi.org/10.1145/3382025.3414973). URL: <https://doi.org/10.1145/3382025.3414973>.



# Contents

List of Figures	v
List of Tables	vii
List of Code Listings	ix
<b>1. Introduction</b>	<b>1</b>
1.1. Challenges for Maneuver-Centric Modeling and Verification . . . . .	2
1.2. Research Questions . . . . .	5
1.3. Approach . . . . .	7
1.4. Reader's Guide . . . . .	9
<b>2. Background</b>	<b>11</b>
2.1. Software Contracts and Contract-based Verification . . . . .	11
2.1.1. Software Contracts . . . . .	11
2.1.2. Contract-based Verification . . . . .	13
2.1.3. Correctness-by-Construction . . . . .	15
2.2. Hybrid Systems Modeling and Verification . . . . .	17
2.2.1. Hybrid Programs . . . . .	18
2.2.2. Differential Dynamic Logic ( $d\mathcal{L}$ ) . . . . .	22
2.3. Assume-Guarantee Reasoning . . . . .	26
<b>3. A Formal Foundation for Skill Graphs</b>	<b>29</b>
3.1. Elements of Skill-Based Modeling . . . . .	30
3.1.1. Requirements Elicitation . . . . .	33
3.1.2. Running Examples . . . . .	34
3.2. Modeling Computation . . . . .	38
3.2.1. A Program Notation for Skills . . . . .	39
3.2.2. Hybrid Mode Automata . . . . .	42
3.3. Syntax and Semantics of Skills and Skill Graphs . . . . .	54
3.3.1. Formalization . . . . .	54
3.3.2. Composition of Skill Graphs . . . . .	59
3.4. Modular Verification of Skill Graphs . . . . .	61
3.4.1. Transformation to Differential Dynamic Logic ( $d\mathcal{L}$ ) . . . . .	61
3.4.2. Compositional Verification . . . . .	66
3.5. Case Study: Vehicle Follow Mode . . . . .	67
3.5.1. Open-Source Tool Support . . . . .	67
3.5.2. Setup . . . . .	69
3.5.3. Results and Insights . . . . .	71
3.5.4. Threats to Validity . . . . .	72

3.6. Discussion . . . . .	72
3.7. Related Work . . . . .	74
3.8. Chapter Summary . . . . .	77
<b>4. Virtual Prototyping of Skill-Graph Maneuvers</b>	<b>79</b>
4.1. Overview of the Verification and Validation Pipeline . . . . .	80
4.2. The ARCHICORC Component Model . . . . .	82
4.2.1. Interface Definition . . . . .	84
4.2.2. Contract Refinement . . . . .	86
4.2.3. Component Definition and Composition . . . . .	87
4.2.4. Excursion: ARCHICORC Code Generation in JAVA . . . . .	90
4.2.5. Discussion . . . . .	95
4.3. Tool Support for Virtual Validation of Skill Graphs . . . . .	98
4.3.1. Automating ARCHICORC Component Generation . . . . .	99
4.3.2. Example: Interfacing with AirSim . . . . .	105
4.4. Evaluation . . . . .	106
4.4.1. Case Studies and Setup . . . . .	106
4.4.2. Results and Insights . . . . .	108
4.4.3. Threats to Validity . . . . .	111
4.5. Related Work . . . . .	113
4.6. Chapter Summary . . . . .	116
<b>5. A Study on Mutation Analysis for Software Contracts</b>	<b>117</b>
5.1. Motivating Example . . . . .	118
5.2. A Taxonomy for Incomplete Specifications . . . . .	120
5.2.1. A Definition of Contract Strength . . . . .	120
5.2.2. A Classification of Incomplete Contracts . . . . .	122
5.3. Mutation Analysis for Software Contracts . . . . .	125
5.3.1. Overview of the Mutation Analysis . . . . .	125
5.3.2. Mutation Operators . . . . .	127
5.3.3. Three Software Metrics for Contract Incompleteness . . . . .	130
5.3.4. Soundness and Completeness . . . . .	131
5.4. Evaluation . . . . .	132
5.4.1. Prototypical Implementation . . . . .	132
5.4.2. Methodology and Evaluated Projects . . . . .	133
5.4.3. Results and Insights . . . . .	135
5.4.4. Threats to Validity . . . . .	141
5.5. Related Work . . . . .	142
5.6. Chapter Summary . . . . .	144
<b>6. GUIDO: Guiding Developers in Configuring Deductive Program Verifiers</b>	<b>147</b>
6.1. Problem Statement . . . . .	148
6.2. Overview of GUIDO . . . . .	151
6.2.1. Configurable Verification Systems . . . . .	151

6.2.2.	Statistical Hypothesis Testing . . . . .	152
6.2.3.	GUIDO Workflow . . . . .	153
6.3.	A Data-Driven Framework for Automatic Configuration . . . . .	154
6.3.1.	Offline Training: Data Set Acquisition and Hypothesis Testing . . . . .	154
6.3.2.	Online Configuration Search as an Optimization Problem . . . . .	159
6.3.3.	Summary of Main Algorithm . . . . .	161
6.4.	Open-Source Implementation . . . . .	163
6.5.	Illustrative Application on Deductive Program verification with KeY . . . . .	164
6.6.	Evaluation . . . . .	165
6.6.1.	Methodology and Evaluated Projects . . . . .	166
6.6.2.	Results and Insights . . . . .	167
6.6.3.	Threats to Validity . . . . .	174
6.7.	Related Work . . . . .	175
6.8.	Chapter Summary . . . . .	177
<b>7.</b>	<b>Conclusion</b>	<b>179</b>
7.1.	Contribution . . . . .	179
7.2.	Future Work . . . . .	180
<b>Appendix</b>		<b>185</b>
<b>A.</b>	<b>Results of the Mutation Analysis</b>	<b>185</b>
<b>B.</b>	<b>Supplemental Material for GUIDO</b>	<b>189</b>
B.1.	Hypotheses for KeY-2.7.0 . . . . .	189
B.2.	Hypotheses for CPACHECKER-4.9.0 . . . . .	191
<b>Bibliography</b>		<b>193</b>





# List of Figures

1.1. Schematic depiction of the control cycle of an autonomous car. . . . .	2
1.2. Schematic overview of an iterative development life cycle with emphasis on deductive verification and virtual validation. . . . .	7
1.3. Overall workflow of the presented maneuver-centric formal engineering process. . .	8
2.1. Exemplified CORC implementation of the update method of class Account. . . . .	17
2.2. Simplified hybrid system of a vehicle with automatic headway control. . . . .	18
2.3. Excerpt of proof rules supported by KEYMAERA X. . . . .	25
3.1. Excerpt of a skill graph representing a maneuver to keep distance to a leading vehicle. .	31
3.2. Illustration of a 2D robot with safety behavior for stationary obstacles. . . . .	34
3.3. Diagram of the modeled skill graph of the Explore World maneuver and its internal structure. . . . .	35
3.4. Hybrid automaton for a thermostat system as presented by Alur et al. [1995]. . . . .	37
3.5. Visual skill-graph diagram of the thermostat system and informal description of its behavior. . . . .	38
3.6. Type hierarchy for skill graphs. . . . .	57
3.7. Translation of hybrid mode automata to hybrid programs. . . . .	62
3.8. Translation of programs in SL to hybrid programs. . . . .	63
3.9. Schematic overview of SKEDITOR. . . . .	68
3.10. Combined skill graph expressing the overall follow mode. . . . .	69
4.1. Schematic overview of steps in the proposed verification and validation pipeline. . .	80
4.2. Schematic workflow of ARCHICORC's development process. . . . .	83
4.3. Transformation of skill graphs to ARCHICORC components. . . . .	98
4.4. Number of top-level modes (behavior) and nondeterministic assignments per case study. . . . .	109
4.5. Measurements for the <i>Adaptive Cruise Control</i> case study. . . . .	111
4.6. Measurements for the <i>Safe Halt</i> case study. . . . .	112
5.1. Venn-diagram illustrating the difference in <i>completeness</i> of two specifications for diverging implementations. . . . .	121
5.2. Taxonomy of causes for incomplete software contracts for object-oriented programming languages. . . . .	123
5.3. Schematic workflow of our mutation analysis for contract-based software. . . . .	125
5.4. Architecture of the prototypical implementation of our mutation analysis framework. .	133
5.5. Percentage of identified incomplete specifications compared to the ground truth. . .	138
5.6. Effectiveness of source-code mutation operators. . . . .	139

5.7. Number of contract-level mutants per project partitioned into killed and alive mutants resulting from operators PW and PS. . . . .	141
6.1. Number of applicable configurations for the program verifier KEY. . . . .	149
6.2. Schematic workflow of GUIDO's offline training phase and online configuration search. . . . .	153
6.3. Exemplified cost graph representing three accepted hypotheses. . . . .	159
6.4. Percentage of verified tasks and compared efficiency for different values of $\gamma \in [0, 1]$ . . . . .	168
6.5. Percentage of closed proofs for the three different continuation strategies. . . . .	170
6.6. Performed verification attempts and accumulated effort for GUIDO and a trial-and-error strategy. . . . .	171
6.7. Reduction in proof size for 14 verification tasks when applying GUIDO instead of the try-and-error strategy. . . . .	172
6.8. Compared efficiency for GUIDO and the default configuration of CPAchecker-1.8 on seven benchmarks. . . . .	173

# List of Tables

3.2.	Excerpt of safety requirements for five hybrid skills of the vehicle follow mode. . . .	70
3.3.	Comparison of the verification effort for skill-based compositional verification. . . .	71
4.1.	Excerpt of code Translation from ARCHICORC components to correct-by-construction JAVA code. . . . .	93
4.2.	Simulation results. . . . .	110
5.1.	List of available source-code and contract-level mutation operators. . . . .	128
5.2.	List of contract mutation rules as proposed by Hou et al. [2007]. . . . .	130
5.3.	Evaluated open-source JML projects. . . . .	134
5.4.	Results of our mutation analysis per project. . . . .	135
6.1.	Set of language constructs $\mathcal{L}$ identified by a source code and specification analysis. .	156
6.2.	Applied significance tests for kinds of hypotheses. . . . .	157
6.3.	Statistics on the Application to KEY . . . . .	164
6.4.	Corpus of JML projects and their characteristics used for evaluating GUIDO on KEY- 2.7.0. . . . .	166
6.5.	Statistics on successfully verified tasks for each continuation mechanism. . . . .	169
A.1.	Mutation scores per method and project for source-code mutation. . . . .	186
A.2.	Mutation scores per method and project for contract-level mutation. . . . .	188
B.1.	Formulated Hypotheses and Experiments . . . . .	191
B.2.	Formulated Hypotheses and Experiments . . . . .	191



# List of Code Listings

2.1.	An Account implementation with contracts in JML. . . . .	12
2.2.	Hybrid program of the automatic headway control. . . . .	21
2.3.	Hybrid program of the automatic headway control with a contract specified in $d\mathcal{L}$ . . . . .	23
3.1.	Time-triggered hybrid program of the <i>explore world</i> maneuver. . . . .	36
3.2.	Excerpt of the textual representation of skill On. . . . .	55
3.3.	Initial version of skill Heater. . . . .	56
4.8.	A thermostat implementation generated by ARCHICORC. . . . .	95
4.9.	C++ implementation of closed-loop control for a lane keeping assistance using AirSim. . . . .	105
5.1.	Excerpt of class LogRecord from the <i>Paycard</i> case study to illustrate incomplete specifications. . . . .	119
5.2.	Json-request for mutating a method from the BankAccount case study. . . . .	133
5.3.	Underspecified method from class OpenJML.Time. . . . .	136
5.4.	Alive mutant of the dutch national flag algorithm as presented by Kourie et al. [2012]. . . . .	137
6.1.	Two methods specified with contracts in JML. . . . .	150
6.2.	Excerpt of domain-specific language for organizing hypotheses. . . . .	163



# 1. Introduction

Cyber-physical systems are ubiquitous in many products that we use in our daily lives, including avionic systems [Sampigethaya et al. 2013], automobiles [Goswami et al. 2012], robotics [Michniewicz et al. 2014], and even medical equipment [Lee et al. 2012]. In his foundational introduction, Alur [2015] defines a cyber-physical system as “*a collection of computing devices communicating with one another and interacting with the physical world via sensors and actuators in a feedback loop*”. This definition captures the hybrid nature of cyber-physical systems, whose behavioral dynamics are both discrete *and* continuous; they integrate digital computation realized through discretely-operating software components with physical processes running in continuous time. For example, cars with an advanced driver assistance constitute cyber-physical systems, whose actuators are discretely controlled by on-board *software* (e.g., setting the acceleration based on sensing the environment), while the car itself is in physical and continuous *motion*.

The question one may ask is whether today’s software development processes for such complex systems are sufficient enough to justify their deployment. Although there exist high safety standards that are imposed on designing these systems (e.g., IEC 61508 [IEC 61508 2011], ISO 26262 [ISO 26262 2011], and ISO 21448 [ISO 21448 2011]), recent reports of fatal accidents with autonomous cars [Poland et al. 2018; Barth et al. 2020; Rice 2019] make clear that shortcomings exist. In 2016, *The Guardian* reported that Google’s autonomous vehicles tested on the roads in California in 2015 suffered 341 disengagements [Harris 2016]. In 69 of these disengagements, the test drivers took rightfully control of the car on their own accord due to mistrust in the car’s exhibited behavior. Even worse, not only are cyber-physical systems difficult to engineer *correctly* in the first place (e.g., due to incomplete requirements and design complexity), but high competition among companies increases the pressure for shorter release cycles, often at the expense of quality assurance [Khomh et al. 2015].

In line with the vision of *verification-driven engineering* [Kordon et al. 2008; A. Müller et al. 2020], a key challenge in developing cyber-physical systems is to integrate formal methods with *model-based design* [Schmidt 2006; Brambilla et al. 2017], the de facto standard in cyber-physical systems engineering. It has been emphasized many times in the literature [Alur 2015; Mitra 2021; Doyen et al. 2018; Platzer 2018; Tabuada 2009] that formal verification techniques need to be applied in this domain to ensure that these systems behave *safely* (i.e., that nothing bad will happen) with respect to a set of safety requirements. Platzer [2018] even argues that “[...] *cyber-physical systems deserve [mathematical] proofs as safety evidence*” to *guarantee* that they interact correctly with the physical world. Additionally, it is not only important to verify conformance to safety requirements, but also to *validate* [Pace 2004] that the modeled behavior fulfills all requirements that are necessary for the intended use-case at run-time (e.g., that the requirements are *complete* enough to operate safely). Whereas ISO 26262 [ISO 26262 2011] focuses more on software engineering processes for ensuring functional safety of the overall systems, ISO 21448 (SOTIF) [ISO 21448 2011] is specifically designed to focus on risk minimization, verification, and validation of the indented functionality, such as assistance systems.

As an example, imagine a newly proposed *advanced driver assistance system* for vehicles that autonomously intervenes with vehicle control. Directly evaluating its feasibility in road traffic would require too much effort or not be feasible at all. Consequently, there is a need for rapidly developing *virtual prototypes*<sup>1</sup> that can already be thoroughly evaluated during the design stages. At the same time, existing approaches and on-going research in this domain [Massow et al. 2018; Massow et al. 2020; Roy et al. 2010; Kutluay 2013] still primarily focus on testing and simulation-based evaluation, which can only identify flaws in the prototype, but not certify their absence. Due to the complexity of road traffic scenarios, there is still a considerable chance of misbehavior as seen in the Google case. This example highlights that new methodologies have to be developed that combine verification at scale (i.e., bringing strong guarantees to large and critical systems to reduce the testing complexity) with virtual prototyping at scale that emphasizes (i) *reuse* of already developed functions and (ii) modeling abstraction. At the same time, we argue that formal verification for functional behavior must be applied at each stage in the development process (i.e., in the design phase, at architectural level, and at implementation level) to comply with these high standards. For us, this raises the question: *what will it take to support developers of all skill levels in formally proving the absence of dangerous defects in their implementations?*

## 1.1. Challenges for Maneuver-Centric Modeling and Verification

Thinking of *maneuvers*, one might first imagine vehicles in road traffic performing well-defined actions, such as *lane changing* or *overtaking*. In this thesis, we are interested in a broader definition of that term, where we roughly define a maneuver as *an externally exhibited behavior of a cyber-physical system that transitions the system from one well-defined state to another well-defined state in the real-world through a cycle of sensing, perceiving, planning, controlling, and acting*. Examples consist of cars with their various driver assistants, but also collision avoidance for marine vehicles, or landing of drones. Figure 1.1 illustrates the computational pipeline for an autonomous car. The specific maneuver (i.e., exhibited behavior) is implemented in the *control unit* that relies on the previous phases and eventually controls the available actuators.

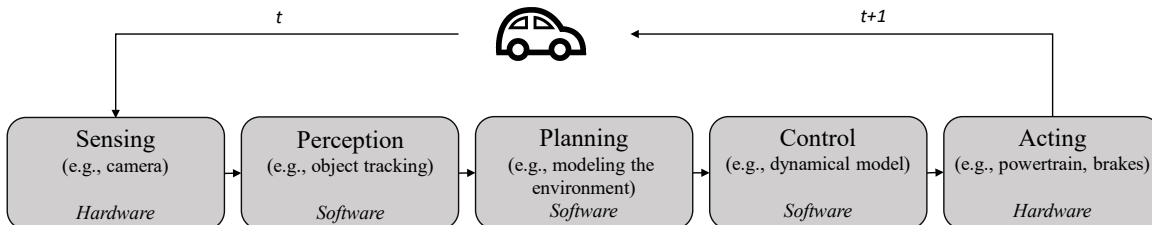


Figure 1.1.: Schematic depiction of the control cycle of an autonomous car.

To ensure that the controller's implementation operates safely, testing is not enough. Most cyber-physical systems that we consider exhibit an almost infinite number of behaviors due to the inter-

<sup>1</sup>Although there exist different definitions, we refer to *virtual prototyping* as a software-based engineering discipline, which carries out modeling, simulation, and visualization under real-world conditions all within a computer. Virtual prototyping is part of the larger concept formed by *digital twins* [Negri et al. 2017].



action with the physical environment [Mitra 2021]. Koopman et al. [2016] argue that “[in] order to validate that the catastrophic failure rate of a vehicle fleet is in fact one per  $10^9$  hours, one must conduct at least  $10^9$  vehicle operational hours of testing (a billion hours)”. Contrary to testing, formal verification mathematically proves that a maneuver  $\mathcal{M}$  does not violate some given property  $\mathcal{P}$ . That is, with formal verification it is possible to guarantee that *all* behaviors of  $\mathcal{M}$  satisfy property  $\mathcal{P}$  and to also *produce* a proof in case of success that gives high assurance guarantees.

In the literature, many languages, techniques, frameworks, and tools exist to formally and semi-formally address the diverse set of challenges of industrial cyber-physical systems development. These challenges include large system sizes, heterogeneity of connected modules, stakeholders from a multitude of disciplines, requirements elicitation, and also software evolution and maintenance themselves. Popular modeling languages include AADL [Feiler et al. 2012; L. Zhang 2013; L. Zhang 2014; Lin et al. 2018; Misson et al. 2019], MODELICA [Elmqvist et al. 2001; Gómez et al. 2020], ALLOY [Jackson 2002; E. Kang et al. 2016], UML [France et al. 1998; Mancini et al. 2018; Jue et al. 2019; Bernardi et al. 2021], and its variants SysML [Friedenthal et al. 2014; Neghina et al. 2020; Pagliari et al. 2020] and MARTE [Bernardi et al. 2007; Seceleanu et al. 2017]. While all these languages greatly contributed to the research of system’s design and analysis, their purpose is (1) to provide rich modeling facilities for almost all parts of a cyber-physical system, and (2) to eventually use these models as basis for real production code. Both goals make these languages inherently complex. For instance, AADL is used to model both hardware and software architectures of real-time embedded systems in great detail. In contrast, we aim to thrive for simplicity and focus on the functional modeling of maneuvers only, which we capture in the following first identified challenge.

**Challenge 1: Modeling Complexity.** Most of the aforementioned modeling languages are too general in nature for our purpose and too complex in their design facilities. This makes it hard for developers to manage their development artifacts and to communicate with stakeholders. Moreover, we argue that higher complexity impairs *reuse* and *modularity* of both the modeling artifacts and analysis results. What we need is a modeling notation that abstracts from unnecessary technical details. The goal should be to start with a simple behavioral model that is already amenable to various analyses – including formal verification – to identify conceptual design flaws as early as possible. Moreover, the modeling notation itself should be platform independent to increase applicability. The main challenge, however, will be the trade-off between simplicity of the modeling approach and substantiality of the early verification results.

Besides modeling languages, there exist a plethora of formal techniques used in the development of cyber-physical systems to enable formal reasoning and verification of these models. Popular techniques include automata theory [Henzinger 2000; Alur et al. 1995; Alur et al. 1992; Lynch et al. 2003; Mitra 2021], abstract state machines [Börger 2010; Metsälä et al. 2017; Drozdov et al. 2019], and system-level contracts [Westman et al. 2013; Nuzzo et al. 2019; Sangiovanni-Vincentelli et al. 2012; Benveniste et al. 2018]. Although contracts in the sense of assume-guarantee reasoning [Benvenuti et al. 2014; Frehse et al. 2004; Henzinger et al. 2001] come with the promise to increase modularity and attack scalability issues, the majority of tools and frameworks integrating contracts applies *model checking* to verify properties of interest. The reason is that these framework aim at verifying more complex and time-related properties, which demands a more expressive specification

language (e.g., linear-time temporal logic [Pnueli 1977] or signal temporal logic [Maler et al. 2004]) and is hard to address with deductive verification. In line with the first challenge, our goal is to focus on safety-related properties (e.g., collision freedom), which we consider to be the most important properties. In the context of deductive verification and relevant for this thesis, there exist *hybrid programs* [Platzer 2008; Platzer 2010; Platzer 2012; Platzer 2018] and *differential dynamic logic* [Platzer 2008; Platzer 2010; Platzer 2012; Platzer 2018], which combine a specialized guarded command language for hybrid systems with a deductive calculus. However, integrating hybrid programs into software engineering processes is still an open challenge. Although A. Müller et al. [2018a] took the first step by combining hybrid programs with component-based design, their approach is still too coarse-grained for our vision of a maneuver-centric modeling and verification approach that maximizes reuse. Thus, the second challenge is as follows.

**Challenge 2: Verification Complexity and Decomposition.** For a scalable verification approach, it is indispensable that a notion of modularity in the process of formal verification exists that allows to verify smaller *functional modules* of the verification model in isolation and uses the correctness results to reason about the correctness of the entire maneuver. It is also of paramount importance to prioritize *reuse* for verification results to decrease human effort, which is best achieved by architectural design. The challenge is to develop a formal concept around verification of safety properties that allows to analyze and verify smaller functional modules, but also allows to reuse such modules automatically when modeling new maneuvers. Ideally, a notion of *compositionality* allows to combine simpler maneuvers to exhibit more complex maneuvers without unnecessary re-verification.

After successful requirement verification, it is also important to *validate* the requirements. This process aims at inspecting completeness of requirements by simulating the modeled behavior using one of many data-driven validation techniques (i.e., run-time verification) [A. Müller et al. 2020]. Unsatisfactory model parts can then be localized and improved. Examples of such simulation and analysis tools include MATLAB/SIMULINK [Angermann et al. 2020], PTOLEMY II [Ptolemaeus 2014], and AADLSIM [Buzdalov et al. 2014]. Again, most of these simulation frameworks focus on richer specification languages, orthogonal problems to functional safety (e.g., uncertainty or performance), or are tied to specific modeling languages. An important challenge we identified is the gap between design model and the actual implementation that is simulated. Most approaches already work with detailed enough design models amenable to simulation (e.g., MATLAB/SIMULINK [Angermann et al. 2020]), whereas our main goal is to maximize abstraction of our designed maneuvers. This requires to *add* details to the implementation model before simulation, which increases the chances to introduce new defects and invalidate simulation results. We formulate the third challenge as follows.

**Challenge 3: Implementation Model and Simulation.** Validating *modeling requirements* demands comprehensive analysis of the maneuver’s observable behavior in a diverse set of scenarios. On the one hand, our goal is to considerably reduce model complexity (see **Challenge 1** and **Challenge 2**). On the other hand, data-driven simulation requires a complete and executable instance of our maneuver. As the verification model of our maneuvers tends to be too abstract for data-driven simulation, we need to bridge the gap between design model and implementation model *without* the introduction of new defects. This demands formal specification and verification at source-code

level to formally cover all parts of the development process.

Finally, our vision is to lower the entry barrier to deductive program verification for developers of all skill levels. Much like software engineering can be seen as the cure to the software crisis, we are now entering an era of *proof engineering* where mathematically proving correctness of real source code resulting in machine-checkable proofs is no longer just a dream (or unbearable labor). Prominent examples of successful verifications include large projects, such as compilers [Leroy 2009; R. Kumar et al. 2014], web browser kernels [Jang et al. 2012], and OS microkernels [Klein et al. 2009; Klein et al. 2014], but also highly-used algorithms in standard libraries [De Gouw et al. 2015; De Gouw et al. 2019; Nipkow et al. 2020]. Moreover, new lines of research, such as automatic proof search [Summers et al. 2018; Bledsoe 2020], proof reuse [Beckert et al. 2004; Bubel et al. 2016], and proof repair [Ringer et al. 2018] acknowledge the role of *developers*, who rarely have the expertise to apply deductive verification interactively.

There exist numerous challenges in the context of deductive program verification that make their adoption in software engineering processes difficult, such as writing specifications [Baumann et al. 2012; Cok 2018], tool integration and reuse [Hähnle et al. 2019], or limited amount of *experimental* research. Recent on-going research [Grebing et al. 2020; Furia et al. 2015; Hiep et al. 2020; Knüppel et al. 2018a] has identified this gap and aims at investigating how to make deductive verification more usable and applicable for practitioners. This thesis makes an effort to contribute to this body of research by also considering how *developers* apply deductive verification when proving correctness of their implementations. Similar to research in software engineering, we argue that combining data-driven techniques (e.g., machine learning) with formal reasoning (1) leads to new observations where new challenges and solutions can be explored and (2) improves integration into software engineering practices. Such efforts include the development of new software metrics for verification projects [Polikarpova et al. 2013; Chockler et al. 2003], conducting empirical studies to improve tool support [Grebing et al. 2020; Knüppel et al. 2018c], or increasing effectiveness of the automated proof search through heuristics [Knüppel et al. 2018b; Knüppel et al. 2018c]. This identified gap results in the final challenge.

**Challenge 4: Mainstreaming Deductive Verification.** Our vision to support developers at all skill levels in their deductive verification projects demands to develop new and *practical* concepts that bridge the gap between software engineering practices and formal reasoning. We believe that it is necessary to (1) increase automation and decrease interaction and (2) increase information and debugging capabilities for developers for both the specification and failed verification attempts. A fruitful direction is to leverage application of deductive verification by connecting this field to other disciplines that prioritize *inductive* learning from data.

## 1.2. Research Questions

Based on the identified challenges above, we formulate the central research question of this thesis as follows.

### Main Research Question

How can we establish a virtual prototyping development process for maneuvers of cyber-physical systems that emphasizes deductive verification throughout and supports developers of all skill levels from model-based design down to source-code level?

For the purpose of this thesis, we narrow this question down to study a specific combination of a graphical modeling notation and a formal framework. In particular, we base our modeling of maneuvers on *skill graphs* [Reschka et al. 2015] combined with a mathematical underpinning by means of hybrid systems [Branicky 2005; Henzinger 2000; Alur 2011], which enables formal verification of the modeled maneuvers. Inspired by ISO 26262 [2011], *skill graphs* [Reschka et al. 2015] have proved to be an effective yet simple starting point in the automotive domain for modeling and organizing new driving maneuvers from an architectural point of view. A skill graph is a directed acyclic graph that visualizes the dependencies between *skills*. A skill is a single functional module that can be categorized into the aforementioned classes *sensor*, *perception*, *planning*, *control*, and *actuator*. Depending on the cyber-physical systems, different skills are available (e.g., different sets of sensors or control algorithms). This decompositional approach maximizes modularity and reuse, as skills can be developed in isolation and combined when necessary.

Hybrid systems are mathematical models that combine discrete and continuous computations, and are thus natural candidates for abstract descriptions of cyber-physical systems. For formally verifying hybrid systems, our focus lies on *deductive reasoning* [Platzer 2010; Zhan et al. 2013; Platzer 2018], where the goal is to logically prove that a given hybrid system satisfies a given set of properties through applying a series of deduction steps (i.e., a *proof*!) with respect to the underlying logical calculus. Most importantly, deductive reasoning allows for compositional verification of hybrid systems, which roughly means that – in principle – it is possible to prove safety of the composition of functional modules solely based on their individual safety guarantees.

From this central research question, we derive the following three more specific research questions, each focusing on a relevant key aspect of our considerations so far.

**Research Question RQ1 – Modeling and Verifying Maneuvers.** *To what extent can a combination of skill graphs and hybrid programs help to model, analyze, and formally prove maneuvers at scale?* As mentioned before, skill graphs allow us to functionally *decompose* behaviors into modules and model their dependencies, and hybrid programs allow us to prove the behavior of cyber-physical systems *compositionally*. Within this research question, we address **Challenge 1** and **Challenge 2**, and investigate the natural thought whether a combination in form of a formal framework is promising for maximizing *reuse* of already developed functions and past verification results.

**Research Question RQ2 – Architectural Refinement and Simulation.** *How can we leverage correct-by-construction component-based design to derive formally correct virtual prototypes amenable to simulation?* To simulate skill graphs, they must be transformed into executable code. Refining skills graphs to component-based architectures is a natural follow-up, as the goal of component-based development is also functional decomposition and reuse. Although the implementation for specific components can be generated automatically while retaining correctness, others need to be developed

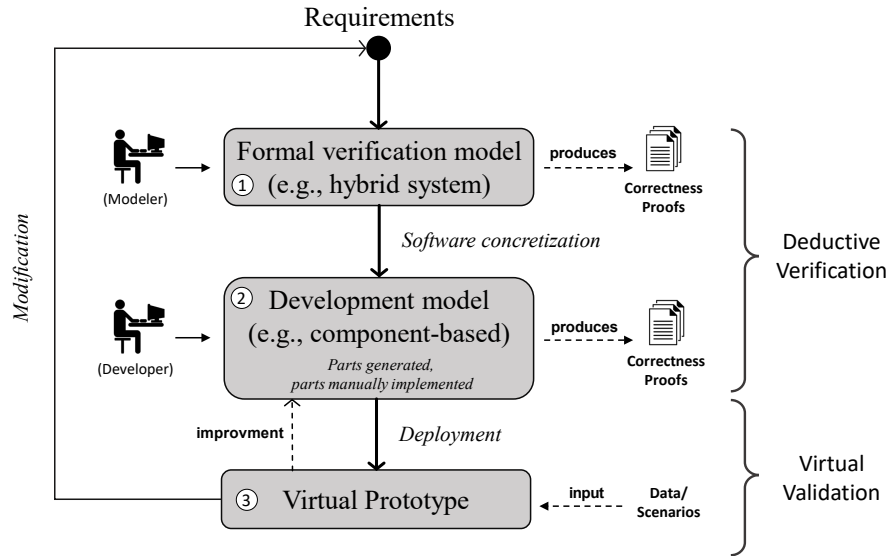


Figure 1.2.: Schematic overview of an iterative development life cycle with emphasis on deductive verification and virtual validation.

from scratch. Within this research question, we address **Challenge 2** and **Challenge 3**, and investigate how we can apply the correctness-by-construction paradigm [Kourie et al. 2012] to derive correct components systematically without introducing new defects.

**Research Question RQ3 – Mainstreaming Deductive Verification.** *In what ways can we support developers to decrease interaction and improve the quality of their deductive verification projects?* Although our focus is on the maneuver-centric verification of cyber-physical systems, there may still exist unverified software parts that can lead to missing bugs and malfunctions. Consequently, a lot of pressure is put on the testing phase after integration (e.g., as suggested by the V model). Ideally, these parts become subject to verification themselves. Our vision is to bring deductive verification into mainstream software development processes. This research question addresses **Challenge 4**. Although numerous challenges still exist [Hähnle et al. 2017; Hähnle et al. 2019], we investigate two challenges that we identified as being of paramount importance, namely (a) assessing the precision of formal specifications and (b) increasing automation of *configurable* deductive verifiers.

## 1.3. Approach

Before we give a more detailed overview of the overall approach and contributions of this thesis, we exemplify our vision for a formal development process emphasizing the application of deductive verification in Figure 1.2. In order to reduce testing effort in the final system, the goal is to apply appropriate means for quality assurance at each development step to quickly handle defects, but eventually to finish the step with correctness proofs by employing deductive program verifiers. These proofs then certify that the current model conforms to the given requirements. After establishing correctness of the behavioral model (i.e., the maneuver) in step ①, it is also nec-



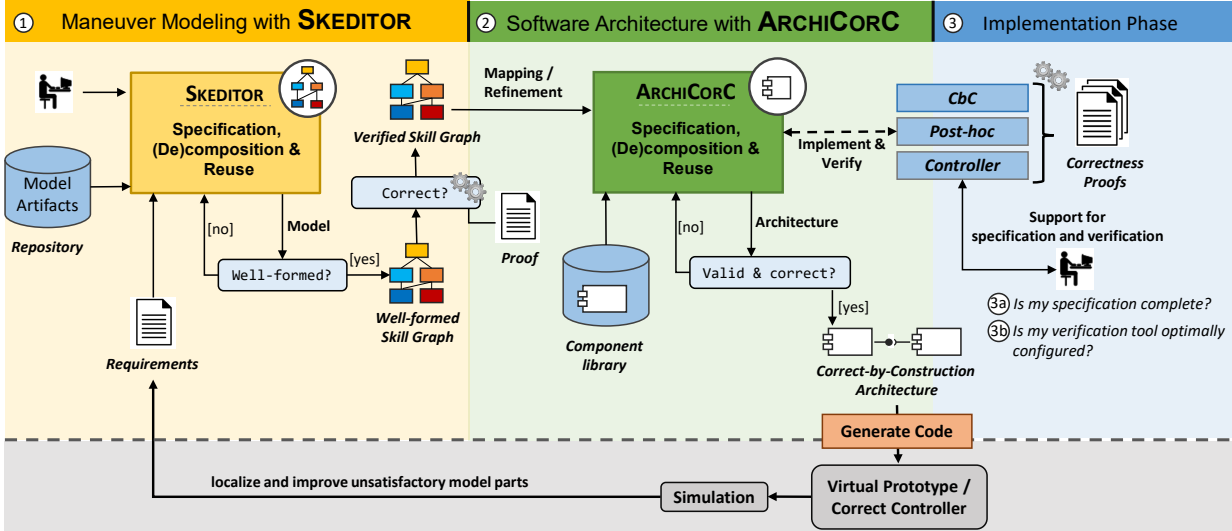


Figure 1.3.: Overall workflow of the presented maneuver-centric formal engineering process.

essary to focus on verification of the executed implementation in step ②. Ideally, each part that is developed manually should be formally verified to mitigate any potential risk of introduced defects or malfunctioning behavior. However, in contrast to the success stories mentioned in the previous section, the integration of formal verification into software development processes is still in its infancy, as it requires high expertise.

As motivated in the previous section, our long-term vision is to allow the virtual prototyping of new and provably correct maneuvers for a variety of cyber-physical systems, and to ease the application of deductive verification in general to support developers and practitioners at all skill levels. Based on the three research questions raised above, this thesis results in four major contributions, each being a stepping stone towards this vision. In Figure 1.3, we show the detailed overview of our approach consisting of the modeling phase, architectural phase, and implementation phase. The numbers ①–③ correspond to the research questions **RQ1–RQ3** of the previous section. In the following, we give a brief description of each contribution.

**Contribution 1: Maneuver Modeling and Verification.** Addressing **RQ1**, we propose a mathematical contract-based framework called **SKEDITOR** for modeling maneuvers of cyber-physical systems and for verifying their safety. Our formalism is based on the notion of skill graphs for modeling and hybrid programs for deductive verification. In particular, we are the first to give a rigorous formalization of skill graphs and present the first tool support for modeling them. We propose *hybrid mode automata* as a formalism to develop skills in isolation, which gives us the properties needed to reduce modeling and verification effort. Based on hybrid mode automata, our framework emphasizes *reuse* and *automatic composition* of already modeled behavior, such that costly re-verification can be prevented. By means of a case study in the automotive domain, we discuss the feasibility of our framework and evaluate to what extent we are able to reduce re-verification.

**Contribution 2: Correct-by-Construction Architecture.** Addressing **RQ2**, we present **ARCHICORC**, a framework that guides the refinement of skill graphs to component-based architectures. While

parts can be generated automatically, safe component behavior of manually developed parts is guaranteed by following the *correctness-by-construction* paradigm. This way, the implemented architecture is also correct-by-construction. ARCHICORC interfaces with visual simulation environments, which allows to validate the initial model requirements. In our evaluation on four non-trivial case studies, we observed that all safety guarantees transferred to the simulated execution model.

**Contribution 3: Mutation Analysis for Software Contracts.** Addressing **RQ3**, we discuss how completeness of contract-based specifications (i.e., preconditions and postconditions) can be assessed with *mutation analysis*, whereas our discussion is based on the specification language JML and the program verifier KEY. We propose a taxonomy of causes for incomplete contracts and propose mutation operators for specification and implementation to identify such causes automatically. By means of open-source JML projects, we evaluate the effectiveness of such an approach.

**Contribution 4: Understanding Parameters of Configurable Program Verifiers.** Again, addressing **RQ3**, we propose a technique to increase the performance and effectiveness of configurable program verifiers by examining the influence of control parameters on the automatic proof search. Our framework, GUIDO, is based on explicit domain knowledge provided by experts that captures assumptions about how control parameters may impact proof search. This formalization of knowledge allows to test assumptions for their validity and to remove false assumptions. In our evaluation, we observed that the one-time effort of formulating the domain knowledge is justified by a significant increase in effectiveness.

## 1.4. Reader's Guide

The order of main chapters follows the order of our four contributions from the previous section. We briefly summarize the four main chapters.

**Chapter 3 – A Formal Foundation for Skill Graphs:** In this first main chapter, we develop a formal concept for specifying and verifying skill graphs based on hybrid mode automata and hybrid programs. All our considerations are implemented in the open-source tool SKEDITOR.

**Chapter 4 – Virtual Prototyping of Skill-Graph Maneuvers:** In the second main chapter, we describe how skill graphs are translated to correct-by-construction component-based architectures based on our framework ARCHICORC.

**Chapter 5 – A Study on Mutation Analysis for Software Contracts:** In the third main chapter, we empirically investigate the effectiveness of a mutation analysis applied on contract-based software.

**Chapter 6 – GUIDO: Guiding Developers in Configuring Deductive Program Verifiers:** In the final main chapter, we introduce our tool chain GUIDO for understanding the influence of control parameters, and for automating parameter tuning in the context of formal verification tools.

Besides these four main chapters, we introduce necessary background in **Chapter 2**. Finally, we conclude this thesis and discuss future work in **Chapter 7**.





## 2. Background

The goal of this chapter is to briefly introduce the key topics of this thesis. In [Section 2.1](#), we introduce *contracts* and *deductive verification* as an abstract means to specify and verify behavior of programs. In [Section 2.2](#), we summarize state of the art on hybrid systems modeling and verification and give a brief overview of *hybrid programs* and *differential dynamic logic* ( $d\mathcal{L}$ ). Finally, in [Section 2.3](#), we present mathematical preliminaries for *assume-guarantee reasoning*, which we rely on throughout the first half of this thesis.

### 2.1. Software Contracts and Contract-based Verification

In this thesis, we primarily focus on *deductive reasoning* techniques to reason about the correctness of program properties, which can be described by means of *contracts*. In [Section 2.1.1](#), we motivate the application of design-by-contract by means of an example. In [Section 2.1.2](#), we characterize the two most prominent approaches for contract-based program verification, namely *theorem proving* and *model checking*. Finally, in [Section 2.1.3](#), we introduce the *correctness-by-construction* paradigm as an alternative to the former two post-hoc verification approaches.

#### 2.1.1. Software Contracts

Long before personal computers became mass-market consumer products, checking the *correctness of programs* was already on the radar of many pioneers in the field of computer science. According to C. A. R. Hoare [\[1981\]](#), none other than Alan Turing – back in 1949 – advocated for using *assertions* (i.e., logical statements that can be proved correct at different locations in a program) to check consistency of program and specification.<sup>1</sup> To follow up on this journey, C. A. R. Hoare [\[1969\]](#) introduced a formal reasoning system, called *Hoare logic*, and a new mathematical notation  $\{P\}S\{Q\}$ , called *Hoare triple*. Hoare triples can be interpreted as follows: *given a program  $S$ , if assertion  $P$  is true before  $S$  is initiated, then assertion  $Q$  will be true after  $S$  is completed*. Nowadays, the terms *precondition* and *postcondition* are commonly used for assertions  $P$  and  $Q$ , respectively.

The term *contract* [\[Liskov et al. 1986\]](#) was initially introduced to refer to pairs of preconditions and postconditions. The *design-by-contract* paradigm [\[Meyer 1992\]](#) generalizes assertions and has proved fruitful in the realm of object-oriented programming. Design by contract was first introduced for the programming language Eiffel, where contracts decorate methods of object-oriented code with a *precondition* and a *postcondition*. Moreover, advanced syntactic sugar was introduced to reduce specification effort, such as *class invariants* or *framing conditions* [\[Hatcliff et al. 2012\]](#). In alignment with the assertional method, preconditions describe what the corresponding method can assume about the

---

<sup>1</sup>In his lecture “Checking a Large Routine” in 1949, Turing made the following comment: “How can one check a large routine in the sense of making sure that it’s right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows” [\[C. A. R. Hoare 1981\]](#).

current state and must be provided by callers. Postconditions describe how the state then changes after executing the method. Class invariants are statements that must always be satisfied during any execution and framing conditions allow to define, which fields a method is allowed to modify.

Contracts formally document the implicit assumptions of programmers explicitly. Depending on the programming language, there exist numerous specification languages with support for contracts. Examples include the aforementioned programming language Eiffel [Meyer 1988] with its inherent support for contracts, SPEC# [Barnett et al. 2011] for C# programs, and the Java Modeling Language (JML) [Leavens et al. 2006], which is a contract-supporting extension of JAVA. In the second half of this thesis, we mainly focus on JML. The reason is that JAVA is one of the most widespread object-oriented programming languages and widely applied in research and industry. Moreover, JML is a mature specification language with a large following.

---

```

1  class Account {
2  public final static int DAILY_LIMIT = 1000;
3  //@ invariant balance >= 0 && withdraw >= 0;
4  public int balance;
5  public int withdraw;
6
7  /*@ requires withdraw < DAILY_LIMIT && amount != 0;
8      @ ensures (\result == \old(withdraw) - amount <= DAILY_LIMIT)
9      @ && (\result ==> withdraw == \old(withdraw) - amount)
10     @ && (\result ==> balance == \old(balance) + amount)
11     @ assignable withdraw, balance;
12     @*/
13  public boolean update(int amount) {
14      if(amount < 0) {
15          if(withdraw - amount > DAILY_LIMIT)
16              return false;
17          withdraw -= amount;
18      }
19      balance += amount;
20      return true;
21  }
22  }

```

---

Listing 2.1: An Account implementation with contracts in JML.

**Example 2.1.** In Listing 2.1, we give an example of a method contract written in JML. Class **Account** stores two fields, namely the current balance and the amount of daily withdrawn money. By calling method **update**, a user is able to deposit or withdraw money. Preconditions in JML are denoted by the keyword **requires**, which assumes that method **update** is only called when the daily withdrawal limit is not yet reached and that the value of argument **amount** is non-zero. Postconditions in JML are denoted by the keyword **ensures**. In this example, the postcondition guarantees that both fields **balance** and **withdraw** are updated according to the provided **amount** if the daily withdrawal limit is not surpassed. Keyword **\old** in a postcondition refers to the state of the

expression prior to method execution and keyword `\result` represents the return value. As illustrated in [Line 3](#), an invariant is defined to prohibit that none of the two fields become negative. Moreover, **assignable** denotes the framing clause [[Hatcliff et al. 2012](#)], which is a set of locations (i.e., fields) that the method is allowed to modify.

Besides class invariants, the second kind of invariant are *loop invariants* [[C. Hoare 1972](#)]. Loop invariants are assertions that are assumed to hold before and after executing the loop. They are often necessary to show termination of `while`- or `for`-loops (i.e., their primary focus is to aid verification). To check that a loop invariant is a correct abstraction of a loop, one has to show that three conditions hold. First, the loop invariant must hold before initiating the loop. Second, it must hold throughout an iteration. Third, it must terminate in a sensible way, which can be achieved by defining a *loop variant* (or bound function) [[Winskel 1993](#)] that monotonically decreases with respect to the loop invariant until a value of zero is reached.

### 2.1.2. Contract-based Verification

In the previous section, we explained how to specify the intended behavior of object-oriented code with software contracts. To verify that method implementations conform to their method contracts, the two most prominent approaches are *theorem proving* [[Schumann 2001](#)] and *model checking* [[Clarke et al. 2018](#)].<sup>2</sup> In the following, we give a brief characterization of these two approaches and how they can be used for contract-based verification.

#### Theorem Proving

*Theorem proving* is a deductive technique for proving the validity of logical formulas. Proofs are carried out in a mathematical style by applying inference rules to formulas in a logic calculus [[Chang et al. 2014](#); [Schumann 2001](#)]. For program verification, a *theorem prover* first translates target programs to logical formulas and postulates theorems about their correctness. Theorem provers then assist programmers with proving these theorems either interactively or automatically, which eventually results in a machine-checkable proof. Interactive theorem provers<sup>3</sup>, such as Coq [[Coq Development Team. The Coq proof assistant, 1989-2021](#)], AGDA [[Agda Development Team. The Agda wiki, 2007-2021](#)], LEAN [[Moura et al. 2015](#)], and ISABELLE/HOL [[Nipkow et al. 2002](#)], embed a *tactics language* representing the inference rules to carry out proofs manually.<sup>4</sup> In contrast, automatic theorem provers, such as VAMPIRE [[Kovács et al. 2013](#)] and SPASS [[Weidenbach et al. 2009](#)], try to synthesize complete proofs (i.e., sequences of inference rules) automatically by employing a proof-search mechanism.

Theorem proving comes with high expressiveness and generality. Automated theorem provers are often restricted to less expressive logic, such as first-order logic, whereas interactive theorem provers focus on higher-order logic. In general, theorem provers allow to encode and prove prop-

<sup>2</sup>Besides theorem proving and model checking, *program analysis* (e.g., different kinds of dataflow analyses and abstract interpretation) is the third prominent category of verification techniques, as acknowledged by the research community [[Clarke et al. 2018](#)]. However, not much emphasis has been put on program analysis and contract-based verification in concert.

<sup>3</sup>Interactive theorem provers are also commonly just referred to as *proof assistants*.

<sup>4</sup>To ease the burden of manually proving every little detail, many modern interactive theorem provers provide means for carrying out simple pieces of a proof automatically (so-called *hammers*). For instance, Coq provides the `auto` and `eauto` tactics.

erties about most problems that arise in the context of verification, including compiler optimization [Leroy 2009; R. Kumar et al. 2014] and contract-based verification of real-world programming languages [De Gouw et al. 2015; De Gouw et al. 2019]. In this thesis, we mainly focus on contract-based program verification, where theorem provers allow to encode semantics of the programming and contracting language precisely (i.e., without the need for abstraction). Deductive program verifiers take a program  $P$  and a contract comprising precondition  $\phi$  and postcondition  $\psi$  as input, and try to solve the following simplified *verification problem*: *if we start in a state that satisfies  $\phi$  and we execute  $P$ , do we end in a state that satisfies  $\psi$ ?* Prominent automated program verifiers with large communities include KEY [Ahrendt et al. 2016] for JAVA/JML, FRAMA-C [Cuoq et al. 2012] for C/ACSL, and VIPER [P. Müller et al. 2016] for numerous front-ends and its own intermediate language.

Proofs are produced following a *proof calculus*, which includes (1) the logical language (e.g., propositional logic or first-order logic), (2) inference rules to prove theorems, and (3) axioms that are assumed to be valid [Wasilewska et al. 2018]. Prominent proof calculi include *Hilbert Systems* [Hilbert et al. 1999], *natural deduction* by Gentzen [1935a], and *sequent calculus* also by Gentzen [1935b]. Sequent calculi are very well studied and integrated into most program verifiers today, as they have practical and theoretical advantages over other calculi; they rely only on very few axioms and provide a rich set of inference rules. In this direction, a *sequent* is of the form  $\Gamma \vdash \Delta$ , where  $\Gamma$  and  $\Delta$  are sets of formulas. The semantics of  $\Gamma \vdash \Delta$  is equivalent to the statement that formula  $\bigwedge_{\phi \in \Gamma} \rightarrow \bigvee_{\psi \in \Delta} \psi$  is a tautology (i.e., always evaluates to *true*). Reasoning is based on *proof rules* that symbolically define which *conclusion* can be drawn from which *premises*. In practice, the goal is typically to start with the conclusion (i.e., the theorem that we want to prove) and to try to find matching premises that are trivially valid.

**Example 2.2.** We give an example of a general proof rule of some sequent calculus. We consider the famous cut rule, which is a generalization of the classical modus ponens and subject of the cut-elimination theorem [Gentzen 1964]:

$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma, \phi \vdash \Delta}{\Gamma \vdash \Delta} \text{ (Cut)}$$

The premises (i.e., sequents above the line) comprise two formulas  $\Gamma \vdash \phi, \Delta$  and  $\Gamma, \phi \vdash \Delta$ , which we assume are tautologies. Then we can draw the conclusion (i.e., sequent below the line) that formula  $\Gamma \vdash \Delta$  is also a tautology. Using symbols  $\Gamma$  and  $\Delta$  as placeholders for arbitrary sets of formulas shows the practical power of the sequent calculus, as it allows to write up a set of such rules concisely and instantiate them in a variety of scenarios as needed.

In practice, theorem provers work with proof scripts that users can inspect manually, but also serve as *certificates of correctness*. This enables advanced techniques that aim to scale theorem proving to large programs by treating proof scripts as programming artifacts. Such techniques include proof reuse [Beckert et al. 2004; Bubel et al. 2016] and proof repair [Ringer et al. 2018]. One of the biggest disadvantages is the needed experience and expertise in proof theory to employ theorem provers effectively, even for automated program verifiers [Knüppel et al. 2018a].

## Model Checking

*Model checking* is a fully-automatic and algorithmic formal verification technique that was proposed independently by Clarke et al. [1981] and Queille et al. [1982] in the early 1980's. Although writing

specifications and employing deductive verification in the style of Hoare was highly appreciated in the eyes of the research community, model checking proved to be most effective in practice for finding programming defects due to its emphasis on automation. Essentially, the verification problem reduces to the question: *given a formal model  $\mathcal{M}$  and a logical property  $\phi$ , does model  $\mathcal{M}$  satisfy property  $\phi$ ?* Model checking answers this question by an exhaustive, yet sophisticated, search procedure. Formal models are often represented by finite automata or – in case of *software model checking* [Clarke et al. 2018] – by software applications. Logical properties are typically written in some temporal logic (e.g., LTL [Pnueli 1977] or CTL [Clarke et al. 1981]), which extends first-order propositional logic by operators that state propositions about time (e.g., that something will *eventually* happen).

There exist a plethora of model checkers nowadays. All have in common that they operate on a particular formal model, as formulated by the verification problem above. For software model checkers, the formal model is often represented by a control flow graph and directly extracted from the input source program. Examples of software model checkers include JAVA PATHFINDER [Havelund et al. 2000] for JAVA programs and CPACHECKER [Beyer et al. 2011] for C programs. Other model checkers operate on dedicated input languages, such as SPIN [Holzmann 1997] operates on PROMELA or NuSMV [Cimatti et al. 1999] operates on the SMV input language. For software model checkers, contracts can be transformed into *run-time assertions* [Hatcliff et al. 2012], which themselves represent the formal properties that have to be satisfied by the program’s execution at explicit locations.<sup>5</sup>

In contrast to theorem proving, most model checkers produce *counterexamples* in case of unsatisfied properties, thereby supporting developers with explanations for the property violations. This contributes to their wide acceptance in practice during development, where debugging is often very time-consuming. However, as model checking is an *automatic* and *exhaustive search* for the violation of logical properties, it is confronted with two major challenges. First, the resources needed for a model-checking task may grow exponentially, such that a model checker may eventually run out of time or memory, which is known as the *state space explosion* problem [Clarke et al. 2018]. Second, the *undecidability theorem* [Turing 1937] rules out the existence of a general sound and complete algorithmic solution for any turing-complete language. To overcome both limitations, software model checkers<sup>6</sup> rely heavily on abstractions for unbound data structures and language constructs, as well as heuristics, which trade precision for decidability and efficiency. In contrast to symbolic model checkers that rely on BDD-techniques, *bounded model checking* is based on SAT solving, and has shown to be more effective and efficient in many experiments that were not solvable with conventional model checking [Biere et al. 1999; Biere et al. 2003]. The automatic proof search is typically tuned by predefined configurations [Beyer et al. 2011].

### 2.1.3. Correctness-by-Construction

In Section 2.1.2, we introduced theorem proving and model checking as means to verify contract-based software. These approaches can be classified as *post-hoc* verification approaches in the sense

<sup>5</sup>Generating run-time assertions from contracts is often performed in the context of test-case generation, where contracts serve as *test oracles* [Hatcliff et al. 2012].

<sup>6</sup>Nowadays, the term *software model checker* is rather misleading, as modern tools are really a culmination of numerous algorithms from three distinct lines of research, namely theorem proving (e.g., SAT and SMT solving), classical model checking, and dataflow analysis.



that the program including the contracts have to be fully written before starting the verification process. Alternatively, *correctness-by-Construction* [Morgan 1994; Dijkstra 1972; Kourie et al. 2012] is a paradigm to derive formally correct programs *incrementally* starting with a specification. That is, correctness of the program is ensured simultaneously to writing the program. Specifications are again Hoare triples [C. A. R. Hoare 1969] written as  $\{\phi\}P\{\psi\}$ , where precondition  $\phi$  defines the initial state, postcondition  $\psi$  asserts the final state, and  $P$  is an abstract statement that is concretized during program construction by applying sound refinement rules. That is, an allowed rule application leads to a primed Hoare triple  $\{\phi\}P'\{\psi\}$ , which is a *refined* (i.e., more concrete) version of  $\{\phi\}P\{\psi\}$  and therefore conforms to the same specification. Eventually, all abstract statements are replaced by concrete program statements and the derived program is guaranteed to be correct by construction.

Recently, CoRC [Runge et al. 2019a] has been proposed, which is a graphical and textual development environment for creating programs following the correctness-by-construction approach. Specifications are written in *first-order logic* similar to JML (i.e., following the *design-by-contract* methodology [Meyer 1992]), and programs are written in an adapted version of the *guarded command language* [Dijkstra 1975]. Incremental derivation of concrete programs from abstract statements is accomplished by the already mentioned *refinement rules*. The six core refinement rules currently supported by the CoRC language are defined as follows.

**Definition 2.1: Core Refinement Rules for the Correctness-by-Construction Approach**

Let  $\phi$  and  $\psi$  be the precondition and postcondition for a program, respectively, and let  $P$  be an abstract statement. Then, the Hoare triple  $\{\phi\}P\{\psi\}$  is **refinable** to

- **Skip:**  $\{\phi\}\text{skip}\{\psi\}$  iff  $\phi$  implies  $\psi$
- **Assignment:**  $\{\phi\}x := E\{\psi\}$  iff  $\phi$  implies  $\psi[x := E]$
- **Composition:**  $\{\phi\}P_1; P_2\{\psi\}$  iff intermediate condition  $M$  exists such that  $\{\phi\}P_1\{M\}$  and  $\{M\}P_2\{\psi\}$  hold
- **Selection:**  $\{\phi\}\text{if } G_1 \Rightarrow P_1 \text{ elseif } \dots G_n \Rightarrow P_n \text{ fi}\{\psi\}$  iff  $\phi$  implies  $G_1 \vee \dots \vee G_n$  and  $\forall i = 1 \dots n : \{\phi \wedge G_i\}P_i\{\psi\}$  holds
- **Repetition:**  $\{\phi\}\text{do } [I, V] G \Rightarrow P \text{ od}\{\psi\}$  iff  $\phi$  implies  $I$  and  $I \wedge \neg G$  implies  $\psi$  and  $\{I \wedge G\}P\{I\}$  holds and  $\{I \wedge G \wedge V = V_0\}P\{I \wedge 0 \leq V < V_0\}$  holds
- **Method Call:**  $\{\phi\}m(a_1, \dots, a_n) \rightarrow r\{\psi\}$  iff method  $\{\phi'\}m(a'_1, \dots, a'_n) \rightarrow r'\{\psi'\}$  exists and  $\phi$  implies  $\phi'[a'_i \setminus a_1]$  and  $\psi'[\text{old}(a'_i) \setminus \text{old}(a_i), r' \setminus r]$  implies  $\psi$

[Runge et al. 2019a]

For the *composition* rule and the *repetition* rule, additional manually written specification is typically required. For the composition rule, an *intermediate condition*  $M$  has to be provided, which evaluates to true after  $P_1$  is executed and needs to be strong enough to serve as the precondition for  $P_2$ . For the repetition rule, a *loop invariant*  $I$  and a variant  $V$  have to be provided. To show termination, the evolution of variant  $V$  needs to be strictly monotonically decreasing with each execution of  $P$  until a lower bound of zero is reached (cf. Section 2.1.1 on *loop invariants*). The *method*

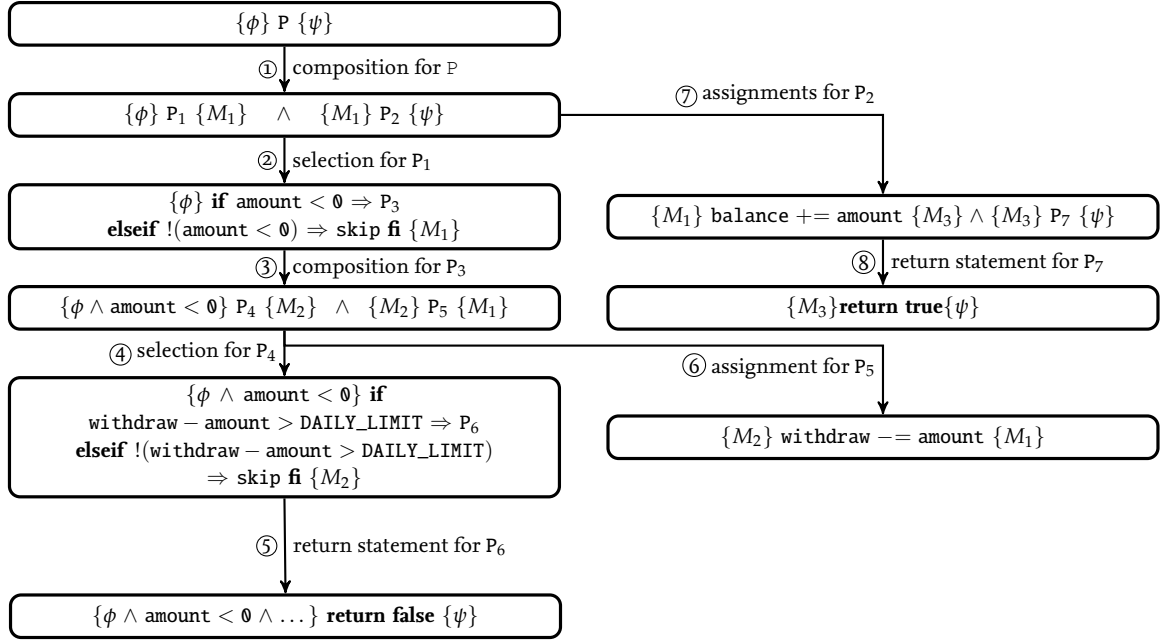


Figure 2.1.: Exemplified CORC implementation of the update method of class Account.

call rule omits side effects (i.e., call by value) and is applicable iff the callee’s specification complies with the respective statement’s specification.

Extending the purely imperative guarded command language, the CORC language introduces a simple object model similar to the one used in ABS [Johnsen et al. 2010]. That is, there exists no inheritance and no subtyping. Moreover, each CORC program is part of an object definition by exhibiting a *method* as declared in classical object-oriented languages (e.g., JAVA). In particular, all referenced variables in a program are labeled as either *argument*, *return value*, or *local variable*. We complete this section with an example of a CORC *diagram* (i.e., a visual representation of a CORC program).

**Example 2.3.** In Figure 2.1, we give an example CORC diagram for the *update*-method introduced in Listing 2.1. Abstract statement  $P$  represents the method itself and is specified with precondition  $\phi$  and postcondition  $\psi$ . To reduce  $P$  to a correct-by-construction implementation, the refinement rules presented in Definition 2.1 are continuously applied until no more abstract statements remain. That is, each leaf node in a complete CORC diagram must end with either an assignment, method call, or skip statement. The circled numbers represent the evaluation order, which is identical to the sequential order of statements given in Listing 2.1. Compared to the design-by-contract paradigm, the difficulty and overhead is in specifying the intermediate conditions (i.e.,  $M_1$  and  $M_2$  in this example). In contrast, blame assignment and debugging become much more precise in practice.

## 2.2. Hybrid Systems Modeling and Verification

To formally reason about cyber-physical systems, they are often mathematically modeled by *hybrid systems* [Branicky 2005; Henzinger 2000; Alur 2011], which mix discrete and continuous behavior in a single formalism and abstract away from unnecessary details. Modeling approaches include *hybrid automata* [Alur et al. 1995] and *hybrid programs* [Platzer 2018]. The following example gives an intuition of the key aspects of hybrid systems.

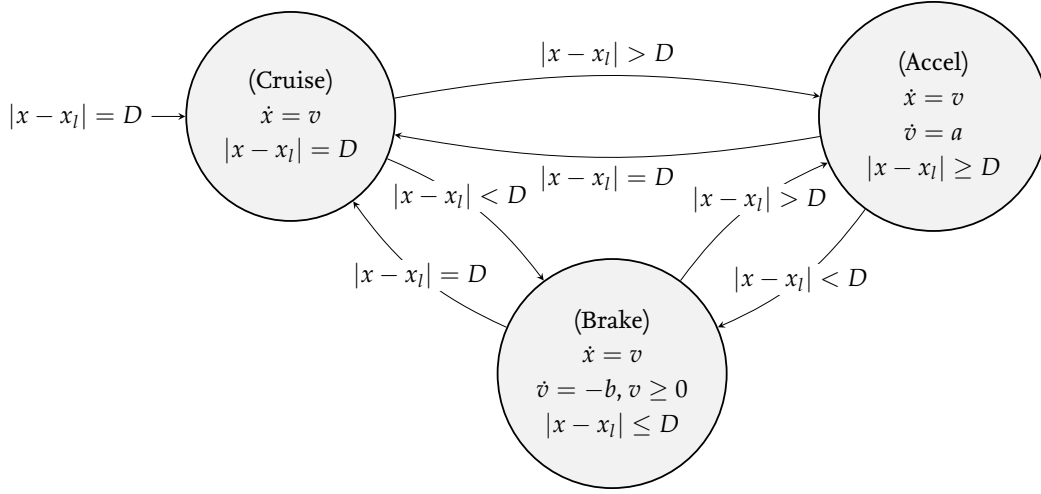


Figure 2.2.: Simplified hybrid system of a vehicle with automatic headway control (adopted from [Knüppel et al. 2020a]).

**Example 2.4.** In Figure 2.2, we give an example of a simplified hybrid automaton representing an automatic headway control with three possible states, namely Cruise, Accel, and Brake. Variables  $x$  and  $x_l$  define the current position on a straight line of both the host vehicle and the leading vehicle, and constant  $D$  represents the ideal distance between them. Moreover, variables  $v$ ,  $a$ , and  $b$  define the current velocity, acceleration, and braking force, respectively. The host vehicle is in cruise mode when the distance to the leading vehicle is equal to distance  $D$ , which also means that acceleration  $a$  is set to zero (i.e., resulting in constant speed). In each control cycle, the current state evolves by applying the respective differential equations and is then evaluated, such that the automaton may transition into a different state. That is, if the leading vehicle increases or decreases the distance, the automaton switches to either state Accel or state Brake. The condition  $v \geq 0$  in state Brake ensures that velocity  $v$  will not be allowed to become negative. To summarize, the headway control ensures that the distance between both vehicles remains approximately equal to  $D$ .

In this thesis, we focus on hybrid programs as the modeling approach for cyber-physical systems, which are closely related to hybrid automata. In Section 2.2.1, we give a brief introduction to the syntax and semantics of hybrid programs. In Section 2.2.2, we describe the ingredients of differential dynamic logic, which enables contract-based deductive reasoning about hybrid programs.

### 2.2.1. Hybrid Programs

Hybrid programs [Platzer 2008; Platzer 2010; Platzer 2012; Platzer 2018] are an imperative-like program notation for modeling hybrid systems. As their purpose is to focus on deductive verification of cyber-physical systems, they extend nondeterministic programming languages [Dijkstra 1975; Floyd 1967] – who share the same purpose of verification for classical algorithms – with differential equations. That is, the classical programming constructs (i.e., *sequential composition*, *assignment*, *nondeterministic choice*, *selection*, *repeat*, *skip*, and *abort*) in a hybrid program represent the algorithmic behavior of a *software controller*, whereas the formulated system of differential equations describes the continuous physical evolution of specific variables in time. Nondeterminism plays an important



role when verifying such programs, as it allows to concisely describe a family of controllers differing in their initial and final state, rather than having one program for every imaginable pair of initial and final state. In the following, we first describe the syntax and semantics of terms and first-order formulas and afterwards the syntax and semantics of hybrid programs.

### Terms and First-Order Formulas

Hybrid programs are primarily concerned with real-valued arithmetic. To provide a rigorous definition of their syntax, we first define how expressions are represented in hybrid programs, which we refer to as *terms* in the remainder of this thesis. The following definition expresses the minimal core of such terms.

#### Definition 2.2: Syntax of Terms

A **term** is defined by the following grammar, where  $\theta_1, \theta_2$  are terms,  $x$  is a real-valued variable, and  $c$  is a rational number constant:

$$\theta_1, \theta_2 ::= x \mid c \mid \theta_1 + \theta_2 \mid \theta_1 * \theta_2$$

[Platzer 2018]

According to **Definition 2.2**, a term is either a real-valued variable  $x$ , a rational constant  $c \in \mathbb{Q}$ , a sum of terms, or a product of terms, where the latter two are defined recursively. Examples for terms are  $x + 1$  or  $x * y + \frac{3}{5}$ . Further useful cases, such as subtraction (i.e.,  $\theta_1 - \theta_2$ ) and division (i.e.,  $\theta_1 / \theta_2$ ) are already included and can be defined in this minimal grammar (e.g.,  $\theta_1 - \theta_2 = \theta_1 + (-1) * \theta_2$ ).

Besides terms, hybrid programs also make use of formulas of first-order logic in combination with real-valued arithmetic. Such formulas are defined as usual and support unary and binary logical connectives, such as *not* ( $\neg$ ), *and* ( $\wedge$ ), *or* ( $\vee$ ), *implication* ( $\rightarrow$ ), and *bi-implication* ( $\leftrightarrow$ ). Additionally, quantifiers *for all* ( $\forall$ ) and *exists* ( $\exists$ ) (quantifying over  $\mathbb{R}$ ) are supported, as well as specific relational operators of real arithmetic for terms, such as *greater-equal* ( $\geq$ ) and *equal* ( $=$ ). The following definition gives a concise summary of the first-order logic used in this thesis.

#### Definition 2.3: Syntax of First-Order Logic of Real Arithmetic

A **formula** of first-order logic of real arithmetic is defined by the following grammar, where  $P, Q$  are formulas,  $\theta_1, \theta_2$  are terms, and  $x$  is a real-valued variable:

$$P, Q ::= \theta_1 \geq \theta_2 \mid \theta_1 = \theta_2 \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q \mid \forall x. P \mid \exists x. P$$

[Platzer 2018]

Reduction of terms and interpretation of first-order real arithmetic as used in this thesis is as usual. For instance, formula  $v \geq 0$  is *true* for all values of variable  $v$  that are greater or equal to zero, and *false* otherwise. The formula  $\exists v(u < v \wedge v < u + 1)$  is valid, as we only con-

sider real-valued variables. For integer variables, the same statement would be *false*, as values between  $u$  and  $u + 1$  cannot be considered.<sup>7</sup>

## Syntax of Hybrid Programs

In [Definition 2.4](#), we give a definition of the EBNF grammar of hybrid programs.

### Definition 2.4: Hybrid Programs

The syntax of **hybrid programs** is defined by the following grammar, where  $\alpha$  and  $\beta$  are hybrid programs,  $x$  is a real-valued variable,  $\theta$  is a term,  $H$  is a first-order logical formula in real arithmetic, and  $x' = f(x)$  is a differential equation:

$$\alpha, \beta ::= \alpha; \beta \mid \alpha \cup \beta \mid \alpha^* \mid x := \theta \mid x := * \mid x' = f(x) \ \& \ H \mid ?H$$

We denote the **universe** of all hybrid programs by HP.

[Platzer 2018]

As mentioned before and visible in [Definition 2.4](#), hybrid programs unify classical constructs of nondeterministic programming with differential equations in one programming model to represent hybrid systems. In the following, we briefly describe the meaning of these constructs in more detail.

**Sequential composition**  $\alpha; \beta$ . Analogous to classical programming, the *sequential composition* starts with the execution of program  $\alpha$  and – after successful termination – continues with the execution of program  $\beta$ .

**Nondeterministic choice**  $\alpha \cup \beta$ . The *nondeterministic choice* provides a behavioral alternative by randomly choosing either to follow the execution of program  $\alpha$  or the execution of program  $\beta$ .

**Nondeterministic repetition**  $\alpha^*$ . The *nondeterministic repetition* expresses that program  $\alpha$  is executed an indefinite number of times (including zero times).

**Discrete assignment**  $x := \theta$ . The *discrete assignment* evaluates term  $\theta$  and instantaneously changes the value of variable  $x$  to the result.

**Nondeterministic assignment**  $x := *$ . The *nondeterministic (discrete) assignment* chooses an arbitrary real value and assigns it to variable  $x$ .

**Dynamic system**  $\{x'_1 = \theta_1, \dots, x'_n = \theta_n \ \& \ H\}$ . The system of *differential equations* expresses a continuous differential evolution of the respective variables  $x_i$  given by the expressions  $\theta_i$ . The evolution is performed nondeterministically for any amount of time but restricted to the *evolution domain* of  $H$ . That is, as soon as condition  $H$  evaluates to **false** in the current state, the continuous evolution terminates. If no evolution domain condition is explicitly provided,  $H$  defaults implicitly to true.

<sup>7</sup>We refer to the second chapter of “*Logical Foundations of Cyber-Physical Systems*” by Platzer [2018] for more information on the semantics of first-order real arithmetic.

**Test condition ?H.** A *test condition* defines a checkable condition that evaluates to either true or false, and may *abort* the execution of the current hybrid program. That is, if test ?H evaluates to **true** (i.e., succeeds) in the current state, no operation is performed (i.e., similar to a `skip` statement in classical programming). If test ?H evaluates to **false** (i.e., fails) in the current state, the current run of the hybrid program is discarded without any change to the current state.<sup>8</sup>

---

```

1  {
2    {
3      ?(dist(x, x_l) = D); a := 0; /* Cruise */
4      U ?(dist(x, x_l) > D); a := A; /* Accelerate with force A */
5      U ?(dist(x, x_l) < D); a := -B; /* Brake with force B */
6    }
7    /* Reset clock timer */
8    t = 0;
9    /* Dynamic System */
10   { x' = v, v' = a, t' = 1 & t ≤ ep & v ≥ 0 }
11 }* /* Repeat arbitrarily often */

```

---

Listing 2.2: Hybrid program of the automatic headway control.

**Example 2.5.** In Listing 2.2, we give an example hybrid program with concrete syntax of the automatic headway control illustrated in Figure 2.2. In each control cycle, the host vehicle chooses a new acceleration  $a \in \{A, -B, 0\}$ , depending on the current distance to the leading vehicle (i.e., represented by the non-deterministic choices and test conditions in Line 3–Line 5). We introduce the helper function symbol  $\mathbf{dist} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$  for computing a comparable distance between the host and leading vehicle (e.g., possibly including model-specific properties, such as the vehicle’s length). A simple version is represented by the absolute difference  $\mathbf{dist}(x, x_l) = |x - x_l|$ . In Line 8, we introduce a clock variable to restrict the continuous evolution of the differential equations to at most  $\mathbf{ep}$  time units (see evolution domain constraint in Line 10), where  $\mathbf{ep}$  can be imagined as the clock period.

### Semantics of Hybrid Programs

To rigorously define the semantics of hybrid programs, we first introduce some necessary mathematical notations as used in related work on hybrid programs. The semantics is based on a *transition relation* between states. We denote by  $\Sigma$  the *set of states*. An element of  $\Sigma$  is a state  $\sigma \in \Sigma$ , which is defined as a function  $\sigma : \mathcal{V} \rightarrow \mathbb{R}$ , where set  $\mathcal{V}$  denotes the set of variables (e.g.,  $x \in \mathcal{V}$  iff variable  $x$  exists). That is, state  $\sigma$  assigns a real (possibly symbolic) value to each variable in  $\mathcal{V}$  (e.g.,  $\sigma(x) = r$  with  $r \in \mathbb{R}$ ). The value of term  $\theta$  as evaluated in state  $\sigma$  is denoted by  $\llbracket \theta \rrbracket_\sigma$ . Moreover, we write  $\sigma \models H$  to express that formula  $H$  is valid in state  $\sigma$ , and we write  $\gamma \models x' = f(x) \ \& \ H$  to express that the *flow*  $\gamma$  (i.e., a timed sequence of states), which depends on the differential equation  $x' = f(x)$ , is always contained within region  $H$ . The semantics of a hybrid program  $\alpha \in \mathbf{HP}$  is then represented as a binary transition relation  $\llbracket \alpha \rrbracket_{\mathbf{HP}} \subseteq \Sigma \times \Sigma$  that specifies which state  $\sigma' \in \Sigma$  can be reached from a state  $\sigma \in \Sigma$ . Formally, the transition semantics is given by Definition 2.5.

---

<sup>8</sup>The test condition ?H is syntactic sugar for **if**  $H$  **then** `skip` **else** `abort fi`, where `skip` has no effect and `abort` discards the current run.

### Definition 2.5: Semantics of Hybrid Programs

The **semantics** of a hybrid program  $\alpha \in \text{HP}$  leads to the following denotational definition of the transition relation  $\llbracket \alpha \rrbracket_{\text{HP}} \subseteq \Sigma \times \Sigma$ , where  $\sigma, \sigma'$  represent the initial and final state, respectively:

- $(\sigma, \sigma') \in \llbracket x := \theta \rrbracket_{\text{HP}}$  iff  $\sigma'(x) = \llbracket \theta \rrbracket_{\sigma}$  and  $\forall y \in \mathcal{V}$  with  $x \neq y$  it follows that  $\sigma(x) = \sigma'(y)$
- $(\sigma, \sigma') \in \llbracket x := * \rrbracket_{\text{HP}}$  iff  $\forall y \in \mathcal{V}$  with  $x \neq y$  it follows that  $\sigma(y) = \sigma'(y)$
- $(\sigma, \sigma') \in \llbracket ?H \rrbracket_{\text{HP}}$  iff  $\sigma = \sigma'$  and the assignment of variables in state  $\sigma$  satisfies formula  $H$  (i.e.,  $\sigma \models H$ )
- $(\sigma, \sigma') \in \llbracket x' = f(x) \ \& \ H \rrbracket_{\text{HP}}$  iff  $\gamma : [0, r] \rightarrow \Sigma$  is a solution with  $\gamma(0) = \sigma$ ,  $\gamma(r) = \sigma'$ , and each state in between  $\gamma(0)$  and  $\gamma(r)$  satisfies formula  $H$  with respect to differential equation  $x' = f(x)$  (i.e.,  $\gamma \models x' = f(x) \ \& \ H$ )
- $\llbracket \alpha \cup \beta \rrbracket_{\text{HP}} = \llbracket \alpha \rrbracket_{\text{HP}} \cup \llbracket \beta \rrbracket_{\text{HP}}$
- $\llbracket \alpha ; \beta \rrbracket_{\text{HP}} = \{(\sigma, \sigma') \mid (\sigma, \sigma_{im}) \in \llbracket \alpha \rrbracket_{\text{HP}}, (\sigma_{im}, \sigma') \in \llbracket \beta \rrbracket_{\text{HP}}\}$  with intermediate state  $\sigma_{im}$
- $\llbracket \alpha^* \rrbracket_{\text{HP}} = \llbracket \alpha \rrbracket_{\text{HP}}^*$  (i.e., the transitive, reflexive closure of  $\llbracket \alpha \rrbracket_{\text{HP}}$ )

[Platzer 2018]

The semantics of the differential system (fourth item) is particularly interesting, as it further highlights the nondeterministic nature of hybrid programs. Imagine, we would omit formula  $H$  in a concrete hybrid program. According to the semantics, *any* reachable state following the defined differential equations could be our final state before the next statement of the program is executed – independent of the time already spent. An evolution domain  $H$  further limits the set of reachable states. This is the exact reason why we modeled a clock period in the form of an evolution domain in [Example 2.5](#), where the set of reachable states reduces to zero once the constraint is unsatisfied. The exact amount of time is still nondeterministic, but we can at least provide a well-defined upper limit.

## 2.2.2. Differential Dynamic Logic ( $d\mathcal{L}$ )

Differential dynamic logic ( $d\mathcal{L}$ ) [Platzer 2008; Platzer 2010; Platzer 2012; Platzer 2018] is a *first-order modal logic* for specifying and proving safety properties of hybrid programs. That is, formulas in  $d\mathcal{L}$  that do not contain modalities are classical first-order logical formulas with real arithmetic. Additionally, the modal operators  $[\alpha]$  and  $\langle \alpha \rangle$  for a hybrid program  $\alpha$  express special reachability properties, which is in accordance with the semantics of hybrid programs that were defined in terms of a transition relation (i.e., reachability of states; see [Definition 2.5](#)). Essentially,  $d\mathcal{L}$  formula  $[\alpha]\phi$  is *true* iff  $\phi$  is true for all reachable states of  $\alpha$ , and  $d\mathcal{L}$  formula  $\langle \alpha \rangle\phi$  is *true* iff  $\phi$  is true in some reachable states of  $\alpha$ . In [Definition 2.6](#), we summarize the grammar of  $d\mathcal{L}$  formulas.

**Definition 2.6: Differential Dynamic Logic  $d\mathcal{L}$** 

The syntax of **differential Dynamic Logic** ( $d\mathcal{L}$ ) is defined by the following grammar, where  $\psi, \phi$  are  $d\mathcal{L}$  formulas,  $\theta_1, \theta_2$  are terms, relation  $\sim \in \{<, \leq, >, \geq, =, \neq\}$ ,  $x$  is a variable, and  $\alpha$  is a hybrid program:

$$\psi, \phi ::= \theta_1 \sim \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \forall x\phi \mid \exists x\phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi$$

We denote the **universe** of all differential dynamic logic formulas by  $d\mathcal{L}$ .

[Platzer 2018]

With formulas in  $d\mathcal{L}$ , it is now possible to specify preconditions and postconditions for hybrid programs in the sense of design-by-contract and Hoare-style deductive reasoning (see [Section 2.1.2](#)). The following example illustrates this.

**Example 2.6.** Consider again [Listing 2.2](#), which represents a hybrid program for an automatic headway control. Surely, our controller should only operate in states that we consider as safe. Consequently, this means that we need to restrict the set of reachable states to safe states (i.e., states in which specific formulas hold). A prominent safety goal that we must ensure for the automatic headway control is collision freedom, where we ensure that the host vehicle does not hit the leading one. This can be expressed in  $d\mathcal{L}$  as  $\psi := x < x_l$ , where  $x$  and  $x_l$  are the position of the host and leading vehicle, respectively. Moreover, a precondition  $\phi$  ensures that we restrict the initial states, such that only safe states are reachable. The verification problem for hybrid programs then becomes: for a hybrid program  $\alpha$ , is formula  $\phi \rightarrow [\alpha]\psi$  valid (i.e., always true)?<sup>9</sup>

```

1  (A > 0 & B > 0 & B ≥ A & v ≥ 0 & ep > 0
2    & x < x_l
3    & x +  $\frac{v^2}{2*B}$  + D ≤ x_l) →  [ {
4    {
5        ?(dist(x, x_l) = D); a := 0; /* Cruise */
6        ∪ ?(dist(x, x_l) > D); a := A; /* Accelerate with force A */
7        ∪ ?(dist(x, x_l) < D); a := -B; /* Brake with force B */
8    }
9    /* Reset clock timer */
10   t = 0;
11   /* Dynamical system */
12   { x' = v, v' = a, t' = 1 & t ≤ ep & v ≥ 0 }
13 }*] /* Repeat arbitrarily often */
14 (x < x_l)
```

Listing 2.3: Hybrid program of the automatic headway control with a contract specified in  $d\mathcal{L}$ .

In [Listing 2.3](#), we extend the previous example with a contract specified in  $d\mathcal{L}$ . We make a number of assumptions. At [Line 1](#), we specify bounds for the used constants and variables. At [Line 2](#), we assume that the host vehicle starts at a position behind the leading vehicle. Finally, at [Line 3](#), we assume that the host vehicle is at a braking distance that is large enough to avoid collision with the leading vehicle. The safety goal  $x < x_l$  stated at [Line 14](#) indeed formalizes this collision freedom between the host and leading vehicle, and must be guaranteed by any possible execution.

<sup>9</sup>Although only implicitly given by [Definition 2.6](#), we use implication ( $\rightarrow$ ) and equivalence ( $\leftrightarrow$ ) where possible.

For the sake of completeness, we give a definition of the interpretation of  $d\mathcal{L}$  formulas in [Definition 2.7](#).

#### Definition 2.7: Interpretation of $d\mathcal{L}$ formulas

The **semantics** (i.e., truth) of a  $d\mathcal{L}$  formula in a state  $\sigma \in \Sigma$  is defined as follows, where  $\phi, \psi \in d\mathcal{L}$ ,  $\alpha \in \text{HP}$ , and we write  $\sigma \models \phi$  to express that formula  $\phi$  is true in state  $\sigma$ :

- $\sigma \models \theta_1 \sim \theta_2$  iff  $\llbracket \theta_1 \sim \theta_2 \rrbracket_\sigma$
- $\sigma \models \neg\phi$  iff  $\sigma \not\models \phi$
- $\sigma \models \phi \wedge \psi$  iff  $\sigma \models \phi$  and  $\sigma \models \psi$  (analogously for  $\vee$  and  $\text{or}$ )
- $\sigma \models \forall x\phi$  iff  $\sigma' \models \phi$  for all states  $\sigma'$ , where  $\sigma'$  agrees with  $\sigma$  except for the values of  $x$
- $\sigma \models \exists x\phi$  iff  $\sigma' \models \phi$  for some state  $\sigma'$ , where  $\sigma'$  agrees with  $\sigma$  except for the values of  $x$
- $\sigma \models [\alpha]\phi$  iff  $\sigma' \models \phi$  for all  $\sigma'$  with  $(\sigma, \sigma') \in \llbracket \alpha \rrbracket_{\text{HP}}$
- $\sigma \models \langle \alpha \rangle \phi$  iff  $\sigma' \models \phi$  for some  $\sigma'$  with  $(\sigma, \sigma') \in \llbracket \alpha \rrbracket_{\text{HP}}$

[Platzer 2018]

#### Free and Bound Variables

To relate the interaction of a hybrid program with its context, we further distinguish between *free* and *bound* variables [Platzer 2017]. Free variables of a hybrid program  $\alpha$  are all those variables that may potentially be read, whereas bound variables are all those that may potentially be written to, either through assignment (e.g.,  $x := *$ ) or differential equations (e.g.,  $x' = v$ ). Importantly, variables can be both bound and free, depending on the content of the program. That is, if there exist an execution, where a variable  $x$  is sometimes written to and sometimes only read,  $x$  is both a free and bound variable. We denote by  $\text{free}(\alpha)$  and  $\text{bound}(\alpha)$  the free and bound variables of hybrid program  $\alpha$ , respectively, and by  $\text{var}(\alpha) = \text{free}(\alpha) \cup \text{bound}(\alpha)$  their union. The sets of free and bound variables can be identified by a static analysis [Platzer 2017].

**Example 2.7.** Consider the following hybrid program:

$$\alpha \equiv (a := 0 \cup a := *; a \leq A); \{x' = v, v' = a \ \& \ v \geq 0\}.$$

Only  $\{A, v\} \in \text{free}(\alpha)$  represent the free variables of program  $\alpha$ , whereas  $\{a, x', v, v'\} \in \text{bound}(\alpha)$  represent the bound variables.  $A$  is only a free variable, as it is only read and never written to in every execution. Conversely,  $a$  is not a free variable, as it is always written to in the controlling part of program  $\alpha$  before being read by the differential equation.

#### Theorem Prover KEYMAERA X

To prove the validity of  $d\mathcal{L}$  formulas deductively, a sound set of axioms and proof rules (i.e., a deductive calculus) is required. The KEYMAERA X theorem prover [Fulton et al. 2015] implements such

$([;]) \quad [\alpha][\beta]\phi \leftrightarrow [\alpha;\beta]\phi$	$(\rightarrow r) \quad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \psi \rightarrow \phi \Delta}$
$([\cup]) \quad [\alpha]\phi \wedge [\beta]\phi \leftrightarrow [\alpha \cup \beta]\phi$	$(\rightarrow l) \quad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \psi \rightarrow \phi \vdash \Delta}$
$([:=]) \quad [x := \theta]\phi(x) \leftrightarrow \phi(\theta)$	$(\wedge r) \quad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}$
$([?]) \quad [?H]\psi \leftrightarrow (H \rightarrow \psi)$	$(\wedge l) \quad \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \phi \wedge \psi, \Delta}$
$(DW) \quad [x' = f(x) \ \& \ H(x)]H(x)$	$(cut) \quad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma, \phi \vdash \Delta}{\Gamma \vdash \Delta}$
$(Wr) \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \phi, \Delta}$	$(Id) \quad \frac{}{\Gamma, \phi \vdash \phi, \Delta}$
$(Wl) \quad \frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta}$	$(ind) \quad \frac{\Gamma \vdash \phi, \Delta \quad \phi \vdash [\alpha]\phi \quad \phi \vdash \psi}{\Gamma \vdash [\alpha^*]\psi, \Delta}$
$([*]) \quad \frac{\forall X[x := X]\phi}{[x := *]\phi}$	$(loop) \quad \frac{\Gamma \vdash \phi, \Delta \quad \phi \vdash [\alpha]\phi \quad \phi \vdash \psi}{\Gamma \vdash [\alpha^*]\psi, \Delta}$
$([M]) \quad \frac{\phi \vdash \psi}{[\alpha]\phi \vdash [\alpha]\psi}$	$(Iter[*]) \quad \frac{\Gamma \vdash \psi \wedge [\alpha][\alpha^*]\psi, \Delta}{\Gamma \vdash [\alpha^*]\psi, \Delta}$
$(CER) \quad \frac{\Gamma \vdash C(Q), \Delta \quad P \leftrightarrow Q}{\Gamma \vdash C(P), \Delta}$	$(CEL) \quad \frac{\Gamma, C(Q) \vdash \Delta \quad P \leftrightarrow Q}{\Gamma, C(P) \vdash \Delta}$

Figure 2.3.: Excerpt of proof rules supported by KEYMAERA X.

a calculus for  $d\mathcal{L}$  and is built on top of a *small trusted kernel* written in SCALA to also increase trust in the tool support itself. Although KEYMAERA X supports a mixture of interactive and automatic theorem proving, its focus is on powerful automation and modular proof management.

KEYMAERA X provides a set of proof rules and axioms, which we will make use of in the next chapter. In Figure 2.3, we present an excerpt of the supported proof rules. The symbols have the same meaning as used before. Proof rules CER and CEL handle *contextual equivalence* to apply the presented axioms for hybrid program transformation. That is, if  $Q \leftrightarrow P$  holds for formulas  $P$  and  $Q$ , then these rules allow to replace any occurrence of  $P$  in any subformula with formula  $Q$  (or vice versa).  $C(\_)$  is used to represent the current context. Among many more, rules for solving or abstracting differential equations in a variety of ways are also supported. A more comprehensive list is given by Platzer [2017].

### Event-Triggered versus Time-Triggered Systems

In principle, there exist two design paradigms for modeling controllers in  $d\mathcal{L}$ , namely *event-triggered design* and *time-triggered design* [Platzer 2018]. Event-triggered control takes action whenever specific events occur. For instance, consider a continuous system  $\{v' = a\}$  with velocity  $v$  and acceleration  $a > 0$ , where our safety goal is to respect a specific speed limit  $v_{max}$ . To achieve this with event-



triggered design, this corresponds to adding a specific evolution domain  $H(v) = v \leq v_{max}$  to the differential equation, resulting in formulas of the form:

$$\phi \rightarrow [(\dots; \{v' = a \ \& \ H(v)\})^*](v \leq v_{max})$$

Based on the semantics of  $d\mathcal{L}$ , the dynamic system may only run as long as the evolution domain is not violated, which *guarantees* the validity of  $v \leq v_{max}$  by design. Such systems are therefore conceptually simple to model and verify, but are difficult to implement faithfully. For instance, events do not consider timing constraints imposed on sensing.

In contrast, time-triggered control takes action sporadically after a certain amount of elapsed time. In  $d\mathcal{L}$ , this corresponds to adding a logical clock  $t$  and a time-bound  $t \leq \text{ep}$  as evolution constraint, where  $\text{ep}$  is the maximum amount of elapsed time between two sensor measurements. The previous example can then be rewritten to :

$$\phi \rightarrow [(\dots; t := 0; \{v' = a \ \& \ t \leq \text{ep}\})^*](v \leq v_{max}).$$

Although time-triggered systems are the only real systems that are implementable, sporadic control based on time is significantly more difficult to model and verify.

## 2.3. Assume-Guarantee Reasoning

Assume-guarantee reasoning [Henzinger et al. 2001; Frehse et al. 2004] is a technique to decompose the verification task into smaller and better manageable subtasks, and is often mentioned in the context of component theories to abstract input-output behavior. In general, such approaches simplify parts of the considered system through abstraction and then show that verifying the simplified parts is enough to proof validity of the original system. Assume-guarantee contracts [Nuzzo et al. 2018; Nuzzo et al. 2015; Benveniste et al. 2007] combine contract theory and assume-guarantee reasoning, and generalize software contracts [Meyer 1992] to contracts for systems design. In Chapter 3, we rely on a notion of *assume-guarantee interfaces* adopted from De Alfaro et al. [2005], which we formally introduce in the following.

### Assume-Guarantee Interfaces

For the sake of presentation, we assume there exists a finite set of variables  $\mathcal{V}$  over which we may define a behavior, and that behaviors can be expressed in terms of variable valuations (i.e., states). We use  $\sigma : \mathcal{V} \rightarrow \mathcal{U}$  with  $\sigma \in \Sigma$  to represent the state at some point in time, where  $\mathcal{U}$  is a typed domain. For the most part of this chapter, we assume  $\mathcal{U} = \mathbb{R}$ , as the focal point of the next chapter will be the deductive verification of the controller logic, where most variables are defined over  $\mathbb{R}$ . Then, these valuations follow the same semantics of state valuations of hybrid programs (see Section 2.2.1). Furthermore, we use symbol  $\mathcal{B}_{\mathcal{V}}$  as an abstraction for an arbitrary implementation (i.e., *behavior*) over variables in  $\mathcal{V}$ , which we can characterize by its externally observable set of executions. Informally, an *execution* is a pair of input and output states, such that each output state is an *update* of the corresponding input state induced by  $\mathcal{B}_{\mathcal{V}}$ .



**Definition 2.8: Execution, Implementation, Trace**

Let  $\mathcal{V}$  be a finite set of variables. An **execution** is a pair  $(\sigma^{\text{in}}, \sigma^{\text{out}}) \in \Sigma \times \Sigma$  of input and output pairs of states. The set of **all executions** restricted to variables in  $\mathcal{V}$  is denoted  $\text{Execs}_{\mathcal{V}}$ . An **implementation**  $\mathcal{B}_{\mathcal{V}}$  is a subset of all executions restricted to variables in  $\mathcal{V}$ , which we denote by  $\llbracket \mathcal{B}_{\mathcal{V}} \rrbracket \subseteq \text{Execs}_{\mathcal{V}}$ . A **trace** of  $\mathcal{B}_{\mathcal{V}}$  is a countable sequence  $\text{trace}(\mathcal{B}_{\mathcal{V}}) = \langle \sigma_1^{\text{in}}, \sigma_1^{\text{out}} \rangle, \dots, \langle \sigma_n^{\text{in}}, \sigma_n^{\text{out}} \rangle$  of executions.

Assume-guarantee contracts bound the exhibited behaviors of an implementation by a set of executions (called *guarantees*) for a preselection of executions (called *assumptions*). The rationale is the same as for software contracts in general, where assumptions and guarantees are described by two predicates (i.e., the precondition and postcondition, respectively).

**Example 2.8.** Consider a postcondition  $\psi_{\mathcal{B}} \equiv \{x \leq u\}$  for a behavior  $\mathcal{B}$  promising that no execution of  $\mathcal{B}$  will include a state pair  $\langle \sigma_i^{\text{in}}, \sigma_i^{\text{out}} \rangle$ , where  $\sigma_i^{\text{out}}(x) > u$  is true, assuming that we start in a state where precondition  $\{x \leq u\}$  is already respected. That is, the following condition is assumed to hold:

$$\forall \langle \sigma_i^{\text{in}}, \sigma_i^{\text{out}} \rangle \in \llbracket \mathcal{B} \rrbracket, \sigma_i^{\text{out}}(x) \leq u. \quad (2.1)$$

In the following, we define *assume-guarantee interfaces*, which simply combine interface theory with assume-guarantee contracts. That is, we explicitly include sets of input and output variables to distinguish between them. As a reminder, we write  $\sigma \models \phi$  to express that formula  $\phi$  is valid in state  $\sigma$  when all free variables are replaced by their respective valuation (see [Section 2.2.1](#)).

**Definition 2.9: Assume-Guarantee Interface**

An **assume-guarantee interface**  $\mathcal{I}$  is a tuple  $\langle V^{\text{in}}, V^{\text{out}}, \phi^A, \phi^G \rangle$ , where

- $V^{\text{in}}$  and  $V^{\text{out}}$  are disjoint sets of input and output variables,
- $\phi^A$  and  $\phi^G$  are two predicates ranging over  $V^{\text{in}}$  and  $V^{\text{in}} \cup V^{\text{out}}$ , respectively.

Furthermore, we define the following two sets of executions induced by  $\phi^A$  and  $\phi^G$ .

- $A_{\mathcal{I}} \subseteq \text{Execs}_{V^{\text{in}} \cup V^{\text{out}}}$  is a set of executions representing the **assumption** with

$$A_{\mathcal{I}} = \{(\sigma^{\text{in}}, \sigma^{\text{out}}) \in \text{Execs}_{V^{\text{in}} \cup V^{\text{out}}} \mid \sigma^{\text{in}} \models \phi^A\},$$

- $G_{\mathcal{I}} \subseteq \text{Execs}_{V^{\text{in}} \cup V^{\text{out}}}$  is a set of executions representing the **guarantee** with

$$G_{\mathcal{I}} = \{(\sigma^{\text{in}}, \sigma^{\text{out}}) \in \text{Execs}_{V^{\text{in}} \cup V^{\text{out}}} \mid \sigma^{\text{out}} \models \phi^G\}.$$

The predicates  $\phi^A$  and  $\phi^G$  are first-order logical formulas over real variables as defined in [Definition 2.3](#). In the following, we briefly define *satisfaction*, *composition*, and *refinement* of assume-guarantee interfaces, as typical for contract-based theories [[Bauer et al. 2012](#); [Westman et al. 2013](#); [Benveniste et al. 2018](#)].

### Satisfaction, Composition, and Refinement

Assume-guarantee interfaces specify the input-output behavior of implementations. As usual, each input covered by the assumption must lead to a state, where the guarantee holds. Then we say that the implementation *satisfies* the assume-guarantee interface. We cover this with the following definition.

#### Definition 2.10: Satisfaction of Assume-Guarantee Interfaces

Let  $\mathcal{I} = \langle V^{\text{in}}, V^{\text{out}}, \phi^A, \phi^G \rangle$  be an assume-guarantee interface and  $\mathcal{B}_{\mathcal{V}}$  an implementation over variables in  $\mathcal{V} = V^{\text{in}} \cup V^{\text{out}}$ .  $\mathcal{B}_{\mathcal{V}}$  **satisfies**  $\mathcal{I}$ , written  $\mathcal{B}_{\mathcal{V}} \models \mathcal{I}$ , if

$$\llbracket \mathcal{B}_{\mathcal{V}} \rrbracket \cap A_{\mathcal{I}} \subseteq G_{\mathcal{I}}.$$

In the context of dynamic logic, we usually write  $\phi^A \rightarrow [\mathcal{B}_{\mathcal{V}}] \phi^G$  with the equivalent meaning.

An important operation on assume-guarantee interfaces is their *composition*, which is possible if two assume-guarantee interfaces can be established in parallel. Two assume-guarantee interfaces are composable – which we refer to as *compatible* – if (1) their output variables are disjoint (syntax), and (2) if the conjunction of their guarantees may imply the conjunction of their assumptions (semantics). We give the following definition.

#### Definition 2.11: Composition of Assume-Guarantee Interfaces

Let  $\mathcal{I}$  and  $\mathcal{I}'$  be two assume-guarantee interfaces.  $\mathcal{I}$  and  $\mathcal{I}'$  are said to be **compatible** iff  $V^{\text{out}} \cap V'^{\text{out}} = \emptyset$  and the following formula  $\psi$  is satisfiable:

$$\psi \equiv \phi^G \wedge \phi'^G \rightarrow \phi^A \wedge \phi'^A.$$

The **composition** is defined as:

$$\mathcal{I} \parallel \mathcal{I}' = \langle V^{\text{in}} \cup V'^{\text{in}} \setminus (V^{\text{out}} \cup V'^{\text{out}}), V^{\text{out}} \cup V'^{\text{out}}, \psi, \phi^G \wedge \phi'^G \rangle$$

[De Alfaro et al. 2005]

Besides composition, the second important operation is *refinement*, which allows to substitute assume-guarantee interfaces. Similar to behavioral subtyping [Liskov et al. 1994], an assume-guarantee interface can be refined to another assume-guarantee interface if (1) the assumptions are only weakened, and (2) the guarantees are only strengthened.

#### Definition 2.12: Refinement of Assume-Guarantee Interfaces

Let  $\mathcal{I}$  and  $\mathcal{I}'$  be two assume-guarantee interfaces.  $\mathcal{I}'$  **refines**  $\mathcal{I}$ , written  $\mathcal{I}' \preceq \mathcal{I}$ , iff (1)  $V^{\text{in}} \subseteq V'^{\text{in}}$  and  $V^{\text{out}} \subseteq V'^{\text{out}}$ , and (2)  $\phi^A \rightarrow \phi'^A$  and  $\phi'^G \rightarrow \phi^G$ .

[De Alfaro et al. 2005]

# 3. A Formal Foundation for Skill Graphs

This chapter shares material with the FASE’20 paper “Skill-based Verification of Cyber-Physical Systems” [Knüppel et al. 2020a].

In the context of (semi-)automated driving, the ISO 26262 [ISO 26262 2011] standard defines *functional safety* concepts for *road vehicles* in general terms for each phase in the development process, namely (1) the requirements analysis (a.k.a. specification elicitation), (2) design, (3) implementation, (4) integration, (5) verification, and (6) validation.<sup>1</sup> Part of the safety life cycle are the *item definition* and *safety goals*. Within the item definition – the very first step in the development process – the to-be-developed system has to be defined in terms of subsystems, functional dependencies, system boundaries (e.g., the environment in which the system operates), intended use case, and some more attributes [ISO 26262 2011]. In a second step, the item definition is then used within a *hazard and risk analysis* to identify hazardous events and to derive safety goals that the system has to satisfy in order to prevent such events. As an example, imagine an automated vehicle that enters moving traffic with a too high steering angle, leading to a high lateral velocity. A hazard and risk analysis may reveal that a potential crash with other traffic participants is possible and conclude that a safety goal limiting the maximum steering angle to a specific degree resolves the issue.

Following the ISO 26262 standard, Reschka et al. [2015] proposed *skill graphs* in the item definition as a means to decompose driving tasks (i.e., maneuvers) into functional units called *skills*. Nolte et al. [2017] give the following abstract definition of a skill.

## Definition 3.1: Skill (Informally)

“A skill describes an activity of a technical system which has to be executed to fulfill the defined goals of the system”.

[Nolte et al. 2017]

Although first ideas for functional system architectures based on *skills* are much older [Maurer 2000; Siedersberger 2004], Reschka et al. [2015] were the first to formulate an *organization scheme* for them in form of a (skill) graph structure. Similar to ISO 26262, skill graphs were intentionally kept vague [Reschka et al. 2015] for general applicability and consequently lack a formal description and semantics. Our goal in this thesis is much more precise: we want to leverage the simple structure of skill graphs to formally (even deductively) verify their maneuver representations at design time.

<sup>1</sup>The ISO 26262 standard itself proposes a development process similar to the classical V-model [Forsberg et al. 1991], but uses vague terminology for the purpose of generalization.

As a byproduct, we support the community centered around skill-based modeling by developing a formal framework and tooling, including a precise formal semantics for skill graphs.

While both concepts – skill graphs and deductive verification of cyber-physical systems – have gained popularity as research topics in recent years, their combination is not straightforward. One key question is how to implement skills and skill graphs, such that their *specification* can be assured; the *ideal* set of assumptions will allow enough flexibility, but at the same time prevent many modeling mistakes upfront. Another key question is how to prevent costly re-verification when skills are reused. We address both questions in this chapter, which is structured as follows. In [Section 3.1](#), we informally describe the ingredients of skill graphs, derive requirements for their formalization, and present two examples for skill-based cyber-physical systems, namely a robot in two dimensions and a thermostat system. In [Section 3.2](#), we present a simple programming model for implementing skills and then present the complete formalization of skills and skill graphs in [Section 3.3](#). In [Section 3.4](#), we demonstrate how the verification problem surrounding skill graphs can be expressed in differential dynamic logic ( $d\mathcal{L}$ ) to verify skill graphs using  $d\mathcal{L}$ 's proof calculus. We also address *modularity* in terms of verifying skills in isolation and only once. We describe our tool support and evaluate our formalization on a case study in [Section 3.5](#). Finally, we discuss fundamental choices we made for our proposed approach in [Section 3.6](#), and discuss related work on functional system architectures, skill graphs, and hybrid system verification in [Section 3.7](#).

### 3.1. Elements of Skill-Based Modeling

Skills closely follow the control cycle of autonomous cars, as depicted in [Figure 1.1](#). Essentially, a skill is an abstract description of something that a cyber-physical system is *capable of* and that can be executed with well-defined inputs and outputs. Skills can be roughly classified into either *sensing*, *perception*, *planning*, *controlling*, or *acting*. Examples in the context of road vehicles include *hardware actuators*, such as the *powertrain* or *brake system*, or *perception algorithms*, such as *perceiving movable objects* through sensors. This categorization follows the design principle of *separation of concerns* [[Parnas 1972](#); [Dijkstra 1976](#)], which prevents modeling mistakes and gives skills well-defined responsibilities.<sup>2</sup> Finally, skills can be annotated with *safety goals* that are elicited by the hazard and risk analysis. These safety goals are invariant, which means that they must be guaranteed to hold initially *and* by any possible trajectory of the represented functionality.

A *skill graph* is a directed acyclic graph that comprises a set of skills (i.e., nodes) and defines dependencies between them (i.e., edges). Conceptually, skill graphs are used for designing and organizing maneuvers of a cyber-physical system. They facilitate the modeling of complex maneuvers built from simpler skills, which interact through explicit *interfaces*. Moreover, they advocate the systematic reuse of ready-to-integrate skills for multiple skill graphs, which reduces maintenance costs and increases software quality in general. Finally, skill graphs are intuitive and therefore accommodate good potential for communicating with stakeholders and non-experts. In [Example 3.1](#), we first discuss a concrete skill graph before describing some of the key aspects in more detail.

<sup>2</sup>Separation of concerns is known to have a positive effect on reducing modeling complexity, increasing comprehensibility, and enabling functional reusability, fault localization, and artifact traceability. Besides that, De Win et al. [[2002](#)] have argued that separating concerns plays also a key role for non-functional requirements.

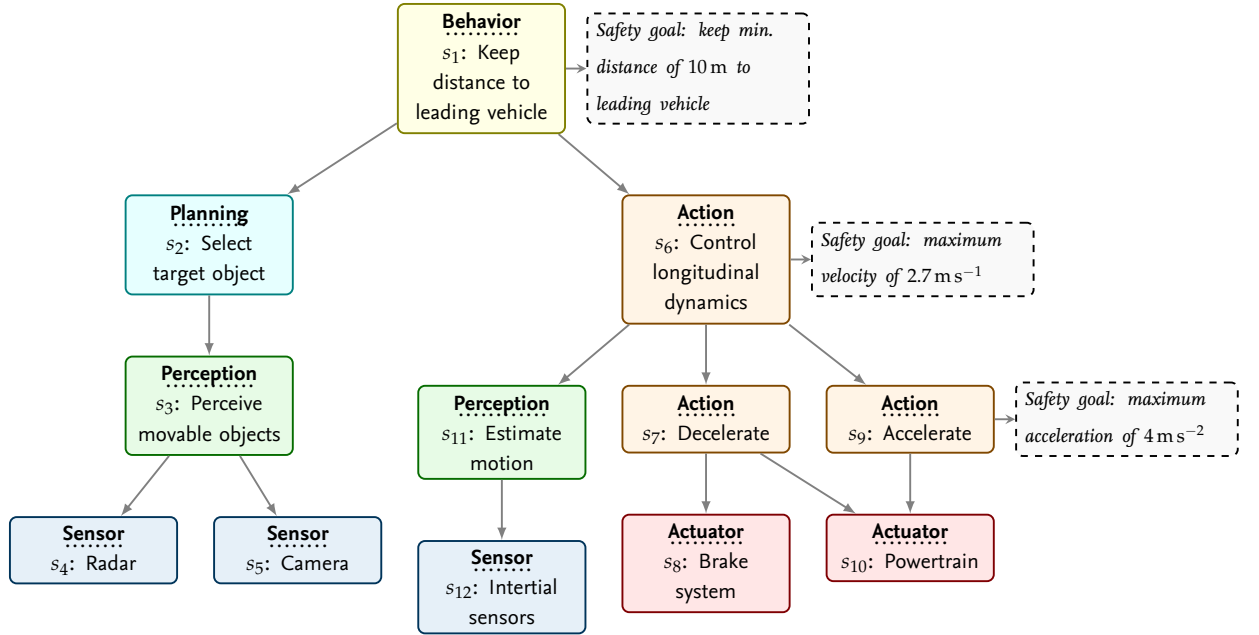


Figure 3.1.: Excerpt of a skill graph representing a maneuver to keep distance to a leading vehicle. We illustrate informal safety goals for the three skills  $s_1$ ,  $s_6$ , and  $s_9$ .

**Example 3.1.** In [Figure 3.1](#), we depict a skill graph representing a driving task, where an unmanned host vehicle tries to keep a distance of at least 10 m to a leading vehicle. On the top level, skill **Keep distance to leading vehicle** ( $s_1$ ) depends on two other skills, namely (1) planning skill **Select target object** ( $s_2$ ) and (2) action skill **Control longitudinal dynamics** ( $s_6$ ). Whereas perception and planning skills are typically realized by software algorithms only (e.g., deep learning for detecting an obstacle), actuator-dependent skills (e.g., action skills) also need to incorporate control theory, as the physical environment has to be taken into account. Skills are annotated with safety goals (e.g., maximum acceleration or minimum distance to other vehicles). Together with the skill's realization and its dependencies to other skills, these requirements express properties that we want to verify at design time. Successfully verifying all skills in the context of a skill graph ensures that the represented maneuver complies to the complete set of safety requirements.

As mentioned before, edges represent dependencies between skills. That is, higher-level skills either depend on information from child skills or can provide input to them. For example, to perceive movable objects, hardware sensors (e.g., cameras) have to provide the raw data to higher-level software skills. Conversely, control skills provide input to (or activate) lower-level skills (e.g., actuators). Edges in a skill graph therefore represent a *requires*-relationship, which explains why both sensors and actuators are at the lowest level; typically for controllers, they take input from sensors and provide output to actuators. Furthermore, as illustrated in [Figure 3.1](#), every path from one skill ends in a *hardware skill* (i.e., sensor or actuator), which itself is not decomposed any further. In the following, we briefly describe the six core types that a skill can have.

**Actuator.** Actuators represent hardware modules tailored to particular cyber-physical systems that allow to control variables. For instance, powertrains for road vehicles allow to control a single variable, namely the acceleration, which influences velocity and position. Other cyber-

physical systems may offer multiple powertrains (e.g., drones; one for each propeller) or only switches that can be turned on or off (e.g., thermal heating systems).

**Sensor.** Sensors represent hardware for data acquisition, such as cameras, radars, or temperature meters.

**Perception.** Perception skills represent software units that interpret sensor data in specific ways (e.g., identifying movable objects). They may prepare such data for planning skills.

**Planning.** Planning skills represent software units that are explicitly concerned with *strategizing and optimizing*, such as computing optimal trajectories or identifying target objects. Although the distinction between *perception* and *planning* is diffuse, it was introduced to explicitly separate both concerns, where perception is rather a subdomain of processing sensor data (e.g., computer vision for image sensors) and planning is rather a subdomain of operations research. Both types of skills may also rely on algorithms rooted in artificial intelligence.

**Action.** Action skills represent *internal* controlling, such as setting the acceleration or activating the brakes. They can be thought of as *spawning subprocesses* that are created or killed by skills of the *observable external behavior* type or other action-typed skills themselves.

**Observable External Behavior.** These skills represent state machines of the actual maneuvers (e.g., keeping the distance to a leading vehicle) as seen from an external observer and are the top-level skills. Technically, they rely on all other skills in a skill graph to form the actual controller logic by taking input data and (de-)activating lower level skills.<sup>3</sup>

On a more abstract level, the *typing* of skills allows to divide them into three distinct categories, namely *hardware skills* (sensor and actuator), *pure software skills* (perception and planning), and finally *hybrid skills* (observable external behavior and action), which mix discrete and continuous dynamics.

An open question is how the underlying behavior of a skill graph is eventually realized to represent the intended maneuver. For instance, Reschka et al. [2015] widely ignore the internal description of skill graphs, as these models are not directly part of an established development process in their research, or not yet linked to implementation artifacts. Foundational work on the predecessor of skill graphs [Siedersberger 2004; Pellkofer 2003], however, proposed to realize skills with *state machines*, where the skill graph's hierarchy represents valid dependencies between the underlying state machines for communication. This way, higher-level skills can invoke state transitions of lower-level skills, similar to hierarchical state machines [Harel 1987; Harel et al. 1998]. States at the lowest level may then access sensors or control actuators. We will adopt a similar notion for realizing hybrid skills, where we refer to states as *modes*.

Our focal point of this chapter will be the development of a sound formal framework that precisely defines how to *construct* and *verify* skill graphs at design time. In Section 3.1.1, we briefly discuss the key ingredients that will be part of our formalism and in Section 3.1.2, we present two informal examples that will paint a clearer picture of how we intend to use and verify skill graphs.

<sup>3</sup>In the remainder of this chapter, we simply use the term *behavior* if the context is clear.



### 3.1.1. Requirements Elicitation

Skill graphs were initially introduced for self-representation and enhanced with *performance measurements* with the goal to check safety goals at run-time through online monitoring [Reschka et al. 2015; Reschka 2017]. In this thesis, we take a step back by focusing on verifying the conformance of skill graph and safety goals at design time, for which a formal foundation of skill graphs is needed, but did not exist before. Following the informal descriptions and considerations of this section, a formalization of skill graphs for the purpose of formal verification must include the following concepts.

- **Formal Description for Skills and Skill Graphs.** Skills in the sense of Reschka et al. [2015] were intentionally kept vague and did not encompass a formal description of their behavior or even guidelines for a thorough implementation. However, skills eventually map to source code in the development process. To analyze maneuvers represented by skill graphs, it is important to develop concepts for modeling their internal structure. As mentioned in the previous section, skills can be realized with a similar formalism to state machines. Another prime aspect of this thesis is formal verification of skill graphs, where a *formal notion of a skill* – including its discrete and continuous effects – is indispensable to reason about their correctness. In summary, hybrid skills have to be associated with a *computational model* (e.g., hybrid automaton) that is abstract enough for ignoring unnecessary details, but rich enough for reasoning about their correctness at design time.
- **Contracts.** Contracts enable assume-guarantee reasoning [Benvenuti et al. 2014; Frehse et al. 2004; Benvenuti et al. 2014] for hybrid systems by abstractly describing the input-output behavior of skills without directly considering the internal structure. The goal is to reason about the correctness of more complex maneuvers by decomposing them into smaller and better manageable subtasks, which are easy to analyze and verify in isolation. Contracts are a natural follow-up to the informal description of skill graphs, where skills are already equipped with safety invariants (i.e., valid regions of variables that must be guaranteed throughout any execution).
- **Modularity and Compositionality.** Finally, the decomposition of the controller logic into various skills requires us to formalize what it means to model complex maneuvers from simpler ones (i.e., by focusing on *reusability*). At the same time, we will study to what extent costly re-verification of already verified parts can be avoided by formalizing the composition of skill graphs that transfers individual safety guarantees to the resulting system behavior automatically.

**Our Approach.** To give skill graphs a precise meaning and to *prove* that they respect their safety requirements, we base our work on automata theory, deductive verification, hybrid programs, differential dynamic logic  $d\mathcal{L}$ , and KEYMAERA X (see Section 2.2). The reason is that deductive methods help us to decompose complex verification tasks into easier manageable subtasks. In contrast, *reachability analysis*, which is the de-facto standard in formal verification of hybrid systems, suffers from the state-space explosion problem, typically requires approximations, and is

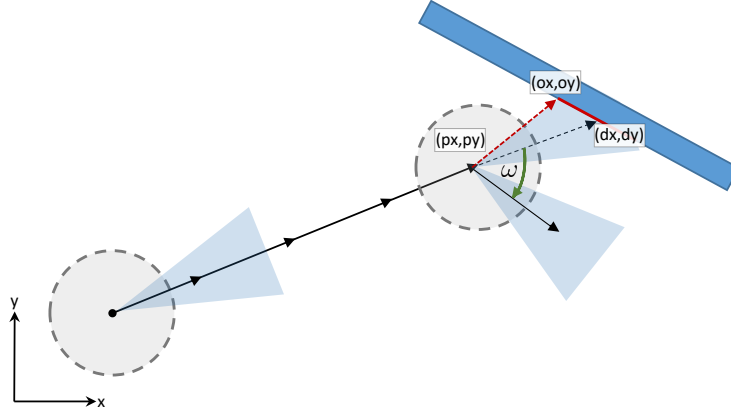


Figure 3.2.: Illustration of a 2D robot with safety behavior for stationary obstacles. The robot must stop (i.e.,  $v = 0$ ) before hitting the wall. Afterwards, it turns right until it is safe to continue driving.

therefore often limited to linear ODEs only. Moreover, tool support for verifying (or even modeling) skill graphs did not exist before. Based on the consideration of this chapter, we introduce the open-source tool SKEDITOR [Knüppel et al. 2020a], which supports users in modeling, analyzing, and verifying their skill graphs in an IDE.

### 3.1.2. Running Examples

Before we formalize skill graphs, we briefly introduce two *simple* examples that we will use to explain our motivation and reasoning for the proposed formalism throughout this chapter. The first example was introduced as part of a bachelor's thesis [Kale 2021] and focuses on a *roomba*-like vacuum cleaner with collision avoidance. For the second example, we model the prominent thermostat system as presented by Alur et al. [1995]. The reason is that – although this thesis places great importance on cyber-physical systems that autonomously move in space (e.g., cars or robots) – this example is simple enough to illustrate many key concepts of our framework, but also showcases that the notion of skill graphs is broadly applicable and may capture a great variety of stationary systems.

#### Example 1: 2D Robot – Explore World Maneuver

Our first example is a *roomba*-like vacuum cleaner, which is a wheeled robot that autonomously moves in a confined two-dimensional area. The robot drives freely, but must avoid collision with any stationary obstacle and border. The robot periodically evaluates the sensor data of an ultrasonic sensor attached to the front and usually moves forward in a straight line. In case of a too close obstacle, the robot *stops* and either turns *left* or *right* (depending on the obstacles position) to identify a new direction without any obstacle in front. As an additional requirement besides avoiding collision, the maximal speed of the robot is limited by a fixed constant  $v_{max}$ . We give an illustration in Figure 3.2, where the robot drives in a straight line with direction  $(d_x, d_y)$  and eventually approaches a wall. The perception of the closest obstacle point  $(o_x, o_y)$  leads to the decision to *stop*, and, after coming to a halt, to *turn right* on the spot until it is safe to accelerate again.



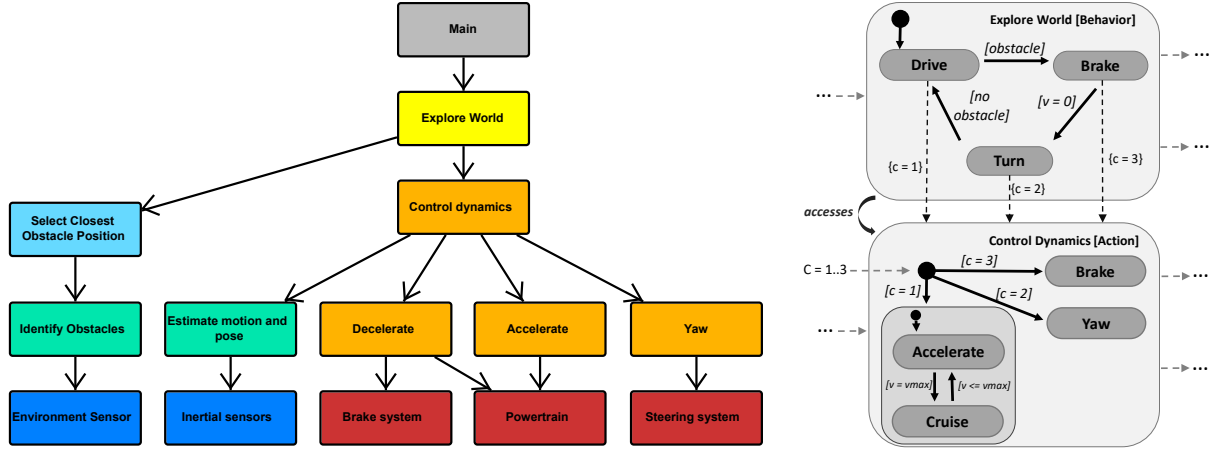


Figure 3.3.: Visual diagram in SKEDITOR of the modeled skill graph of the Explore World maneuver (left) and schematic visualization of the underlying state machines for skills **Explore World** and **Control Dynamics** (right).

**Skill Graph.** We present a possible diagram of a skill graph in Figure 3.3 that models the described behavior. The skill graph comprises an environment sensor (i.e., ultrasonic) and an inertial sensor, which is used to estimate the current position, velocity, and referential direction. The actuators are as usual for mobile robots (i.e., powertrain and brake system for controlling longitudinal movement, and steering system for controlling lateral movement). This emphasizes that each intended class of cyber-physical systems comes with its own set of usable skills.

The actual behavior is modeled in skill **Explore World** (yellow) and can be described with a state machine (see Figure 3.3, on the right). The behavior consists of three top-level modes: [Drive], [Brake], and [Turn]. Initially, the robot is stopped (i.e.,  $v = 0$ ), but directly enters the driving mode. At each control cycle, the sensor values are evaluated and the state machine may transition to a different mode. In the presence of an obstacle, the robot transitions to mode [Brake], remains there until it has completely stopped, and afterwards transitions to mode [Turn], where it turns until a new safe direction is identified.

**Explore World** depends on action skill **Control dynamics** (orange), which itself is a *different* kind of controller. It offers four modes, namely [Accelerate], [Cruise], [Brake] (same name but not the same mode as above), and [Yaw]. However, modes in this case are activated via a control input  $c \in \{1 \dots 3\}$ . [Accelerate] and [Cruise] build a pair to approach a reference velocity  $v_{max}$ , whereas the other two modes are activated in isolation. On a theoretical level, top-level skills are thought to be the *controller* for lower level skills. That is, mode [Drive] of **Explore World** instantiates the pair of modes [Accelerate] and [Cruise] of **Control dynamics**. If an obstacle is close enough, the controller for **Explore World** will transition to mode [Brake]. Analogously, this causes the controller of **Control dynamics** to transition to its mode [Brake] as well.

**Behavior and Safety.** The overall safety goal is to avoid collision at all cost, which we wish to verify given our modeled behavior. For the sake of completeness, we give a formal description of the described behavior as a  $d\mathcal{L}$  formula. We assume that the robot is located at position  $(p_x, p_y)$  and is initially stopped with velocity  $v = 0$ . The goal for the robot is to drive in direction  $(d_x, d_y)$  in a

straight line with maximum speed  $v_{max}$ . If a close obstacle is detected at location  $(o_x, o_y)$ , the robot decelerates *early enough* to avoid collision. This is explicitly modeled by the worst-case execution time  $ep$  that represents the maximum elapsed time between two sensor measurements. Furthermore, we introduce a logical clock  $t$  that represents the *exact* elapsed time between to sensor measurements to discretize the continuous flow of the control variables. The  $d\mathcal{L}$  formula presented in Listing 3.1 expresses the overall behavior and safety goal of this maneuver, where we ignored the concrete implementations for  $\text{safeDist}(v)$  (“is the distance safe to accelerate?”),  $\text{safeAcc}(v)$  (“have we reached the safe maximum velocity?”), and  $\text{chooseSteering}(p, o)$  (“choose an adequate yaw rate!”). Variable  $w$  represents the current steering angle in radians.

```

1  (A > 0 & B > 0 & B ≥ A & v = 0 & a = 0 & w = 0 & ep > 0
2    & ||p - o|| > 0 & dx2 + dy2 = 1) → [ { {
3      /* Mode 1: Drive forward with direction (dx, dy) */
4      ? (safeDist(v) < ||p - o||); {
5        w := 0;
6        /* Mode 1a: accelerate; Mode 1b: cruise; */
7        { ? safeAcc(v); a := A; ∪ ? ¬ safeAcc(v); a := 0; }
8      }
9      /* Mode 2: Distance violated -> brake! */
10     ∪ ? (safeDist(v) ≥ ||p - o|| & v > 0); {
11       a := -B;
12     }
13     /* Mode 3: Find new safe direction */
14     ∪ ? (safeDist(v) ≥ ||p - o|| & v = 0); {
15       a := 0; w := chooseSteering(p, o);
16     }
17   }
18   /* Reset clock timer */
19   t = 0;
20   /* Dynamical system */
21   { x' = v * dx, y' = v * dy, dx' = -w * dy, dy' = w * dx, v' = a, t' = 1 & t ≤ ep & v ≥ 0 }
22 } * ] (||p - o|| > 0) /* Safety inv.: do not collide with any obstacle!! */

```

Driving

Braking

Rotating

Listing 3.1: Time-triggered hybrid program of the *explore world* maneuver with robot position  $p = (p_x, p_y)$  and closest obstacle position  $o = (o_x, o_y)$ .

Besides the goal of giving skill graphs a formal meaning, this example hints at the fact that skill graphs may also give hybrid programs more structure. Although this example is fairly small, formulating more complex maneuvers will definitely benefit from reusing structures and proof results. In the remainder of this chapter, we will see how this modeled behavior in  $d\mathcal{L}$  can be decomposed into skills. These skills can be verified in isolation and composed together to even develop new maneuvers – all while retaining some correctness results.

## Example 2: Thermostat System

In this example, we model a heater with a thermostat controller. The current temperature is denoted by  $x$ . The goal of the controller is to keep the temperature between a lower limit ( $l$ ) and an upper

limit ( $u$ ). If the temperature rises above a specific temperature  $u - \delta_u$ , the heater is turned off. Likewise, if the temperature falls below a specific temperature  $l + \delta_l$ , the heater is turned on again.

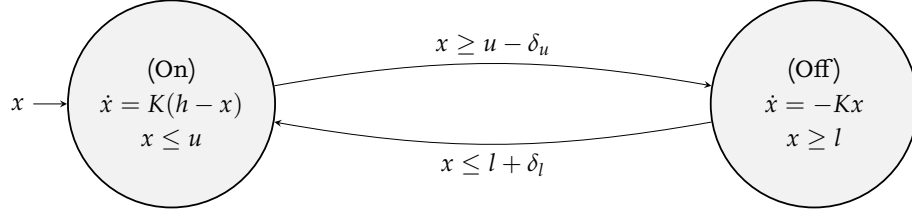


Figure 3.4.: Hybrid automaton for a thermostat system as presented by Alur et al. [1995].

In Figure 3.4, we illustrate the thermostat system as a hybrid automaton. Constant  $K$  is determined by the room temperature, whereas variable  $h$  is the *heating power*. When the heater is off, the temperature follows the function  $x(t) = x_0 * e^{-Kt}$  (i.e., solution of the differential equation), where  $t$  represents time and  $x_0$  represents the initial temperature. Analogously, if the heater is on, the temperature follows the function  $x(t) = x_0 * e^{-Kt} + h(1 - e^{-Kt})$ . The safety margins  $\delta_u > 0$  and  $\delta_l > 0$  play a key role in *time-triggered* systems (cf. Section 2.2.2), where we also consider the passed time between two measurements depending on the sensor frequency. That is, it must be ensured that the heater transits to the other mode before the *next* measurement may reveal a violation of the invariant, even if the current state is relatively safe.

In contrast to the previous example, the discrete controller logic of this system in each individual mode is almost trivial. If the heater is [off], the discrete controller is idling and does not control any variable in its discrete part. If the controller is [on], we may decide to either view  $h$  as a fixed constant or to dynamically adjust  $h$  with respect to the current temperature. In the former case, no variable is changed again in the discrete controller part, which means that the whole system only depends on the dynamic part. It is still worth studying such systems in the context of this work, as this version of a thermostat may only be the first version in a series of incrementally developed and more complex systems.

**Skill Graph.** The illustrated skill graph in Figure 3.5 consists of only four skills (besides a root skill): one sensor (**Temperature**), one actuator (**Furnace**), and two hybrid skills (**On** and **Heater**). The sensor's temperature  $x$  is accessible by all skills on a *path*. Analogously, the furnace skill requires as input the heating power  $h$ , which has to be controlled by hybrid skills on a path. We will later see that both types of hybrid skills (i.e., *action* and *behavior*) are similar in their internal structure and can oftentimes be used interchangeably. The difference is that each skill graph must have at least one behavior skill of and that action skills are always lower on the hierarchy. How skill graphs are modeled therefore varies greatly. For the thermostat system, we decided to model mode [on] in a separate action skill **On**, whereas mode [Off] is modeled as part of skill **Heater**. Alternatively, it is also reasonable to model both modes as part of skill **Heater**, which could reduce reusability, or both modes in separate action skills, which would increase implementation and maintenance effort.

**Behavior and Safety.** We annotated both hybrid skills with a *precondition* (i.e., assumption, under which this skill can be executed) and *postcondition* (i.e., guarantee or safety invariant). For the sake

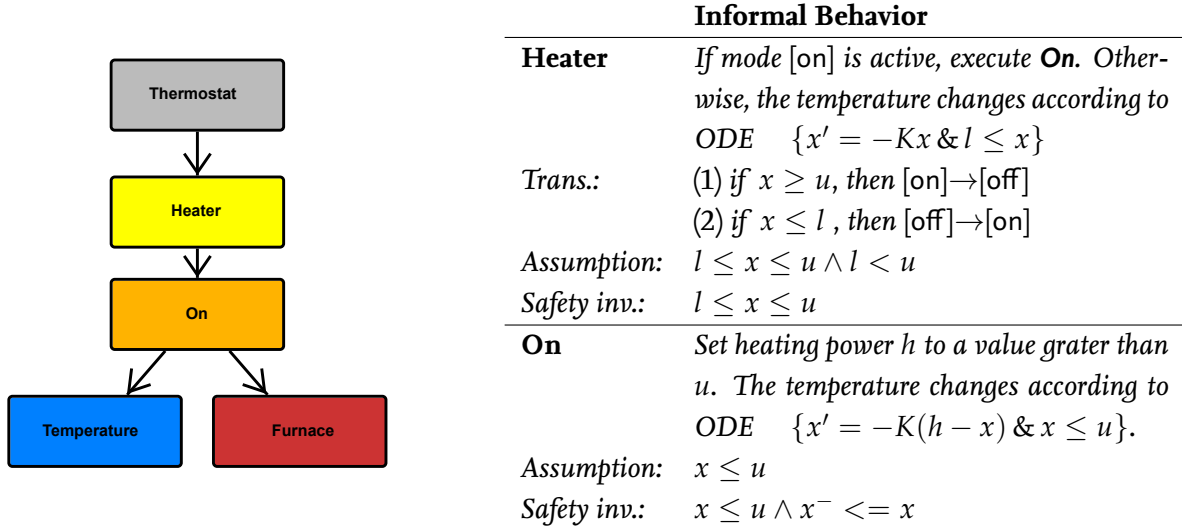


Figure 3.5.: Visual skill-graph diagram of the thermostat system as modeled in SKEDITOR (left) and informal description of its behavior (right).

of simplicity, we modeled the thermostat system in this example as an *event-triggered system*. That is, we assume that  $\delta_u = \delta_l = 0$  and that a transition will always take place before a safety invariant is violated. For skill **Heater**, we require that the currently measured temperature is between a lower and upper bound ( $l \leq x \leq u \ \& \ l < u$ ) and then guarantee that it will not leave that region ( $l \leq x \leq u$ ). Skill **On** is an isolated sub-behavior (similar to procedures) that only focuses on the upper limit  $x \leq u$  in both pre- and postcondition. Moreover, we specify that, in mode [on], the temperature only increases by referring to  $x$ 's past edition  $x^-$ . The behavior **On** is activated and deactivated by skill **Heater**. Consequently, in a verification scenario, it must be ensured that skill **On** can only be activated if it safe to do so (i.e., if  $x \leq u$  holds). The table on the right side of Figure 3.5 paraphrases the informal behavior of both skills. Skill **Heater** serves as the main controller that switches between modes [on] and [off]. Whenever mode [on] is entered, action skill **On** takes over. A transition to [off] is triggered if  $x \geq u$  (i.e., the temperature increased too much). In theory, however, the evolution constraint of the ODE of skill **On** will prevent any scenario where  $x > u$ , such that  $x \leq u$  is guaranteed. As mentioned before, realistic systems are not event-triggered, but time-triggered.

## 3.2. Modeling Computation

As described in Section 3.1, skills in the original sense can be viewed as encapsulations of *some functional behavior*. This view is vague in the sense that it does not give meaning to the concrete *type* of behavior that is used. In particular, a behavior can be described by many formalisms, such as automata, labeled transition systems, differential equations, specifications, and more. We have also seen that skill graphs unite different kind of these behaviors (i.e., hardware, pure software, and discrete controllers with continuous flow). In this chapter, the behavior of hardware and software skills will only be described by their assumptions and guarantees without focusing on the internal structure. In contrast, how to model and verify the *controller logic* represented by hybrid skills

will be discussed in great detail. The goal of this section is to develop a programming model for *hybrid skills* for the purpose of analysis and verification.

As a first step, we derive a concrete program notation for implementing hybrid skills in [Section 3.2.1](#) that is rooted in hybrid programs (see [Section 2.2.1](#)). Thereupon, in [Section 3.2.2](#), we derive a more flexible computational model, namely *hybrid mode automata*, for implementing hybrid skills.

### 3.2.1. A Program Notation for Skills

For describing the behavior of skills, we will closely follow the syntax and semantics of hybrid programs [[Platzer 2008](#); [Platzer 2010](#); [Platzer 2012](#); [Platzer 2018](#)], which itself is only a simple non-deterministic programming language with support for differential equations. The reason is twofold. First, hybrid programs provide a very simple programming model, which is expressively sufficient for us to describe the intended controller logic at this design stage. Second, we aim at leveraging  $d\mathcal{L}$  [[Platzer 2008](#); [Platzer 2010](#); [Platzer 2012](#); [Platzer 2018](#)] for verifying skill graphs. As  $d\mathcal{L}$  is defined over hybrid programs, it is natural to only make some necessary modifications. The language we choose for implementing skills will be denoted SL (for *Skill Language*). We first give a definition of the syntax of SL and afterwards describe the differences to hybrid programs in more detail.

#### Definition 3.2: Syntax of SL

The **syntax** of language SL for implementing skills is defined by the following grammar, where  $S_1$  and  $S_2$  are programs of SL,  $x, x_1, \dots, x_n$  are real-valued variables,  $\theta$  is a term,  $P, Q$ , and  $H$  are first-order logical formulas in real arithmetic,  $M$  is an abstract placeholder, and  $x' = f(x)$  is a system of ODEs:

$$S_1, S_2 ::= S_1; S_2 \mid x := \theta \mid \text{havoc } x_1, \dots, x_n \mid \text{assume } H \mid \text{skip} \mid x' = f(x) \ \& \ H \\ \mid \text{if}(H) \{ S_1 \} \text{else} \{ S_2 \} \mid \text{assert } H \mid \text{invoke } M \mid (P)M(Q)[x_1, \dots, x_n]$$

Although our syntax slightly differs from the one used for hybrid programs, most constructs have the same meaning; besides composition, discrete assignment, and the dynamic system, **havoc**  $x_1, \dots, x_n$  is the equivalent of nondeterministic assignment (i.e.,  $x_1 := *; \dots; x_n := *$ ), **assume**  $H$  is the equivalent of a test condition (i.e.,  $?H$ ), **skip** is shorthand for **assume** true, and the selection statement **if**( $H$ )  $\{S_1\}$  **else**  $\{S_2\}$  is equivalent to the hybrid program  $?H; S_1 \cup ?\neg H; S_2$ . Nondeterministic repetition and nondeterministic choice are not part of our syntax. The reason for the former is that a skill may only perform one execution per (synchronized) time step, which makes repetition unnecessary. The reason for the latter is that nondeterministic choice needs to be resolved when deriving a concrete implementation. We restrict language SL to nondeterministic assignment only, which is simpler to resolve. In the following, we briefly describe the meaning of the additional constructs **assert**  $H$ , **invoke**  $M$ , and  $(P)M(Q)[x_1, \dots, x_n]$ .

**Assertion** (**assert**  $H$ ). An *assertion* is similar to a test condition, where the checkable condition  $H$  evaluates to either true or false depending on the current state. However, a violation of  $H$  will not just abort the current run, but the program will transit to a designated *error state*, which itself does not have outgoing transitions. This way, we mark violations of  $H$  explicitly as erroneous behavior.

**Program invocation (invoke M).** An *invocation* of an SL program represented by the abstract symbol  $M$ , and resolved by some binding  $\zeta(M)$ , simply follows the execution of the statements in  $M$ . An invocation is the equivalence of *inlining*, where  $M$  represents a non-parameterized macro. If we require that  $M$  is specified with a contract (cf. [Section 2.1.1](#)), then instead of inlining  $M$ , we can assert  $M$ 's precondition and assume  $M$ 's postcondition to increase modularity.

**Hoare triple ( $\langle P \rangle M \langle Q \rangle [x_1, \dots, x_n]$ ).** The triple  $\langle P \rangle M \langle Q \rangle [x_1, \dots, x_n]$  expresses that if we can *assert*  $P$ , we may *assume*  $Q$  for all accessible variables  $x_1, \dots, x_n$ . The last part is the *framing condition* [[Hatcliff et al. 2012](#)], which limits the effect of postcondition  $Q$  to specific variables. Similar to nondeterministic assignment, a Hoare triple is in general not executable on a real computer and must be refined to executable statements. However, it helps in the process of incremental development and asserting correctness.

Using invocations and Hoare triples directly in our syntax will give us some more flexibility compared to hybrid programs alone when designing skills in isolation. Invocations allow us to increase modularity and reusability by developing the controller logic over multiple programs. Hoare triples allow us to develop skills and skills graphs incrementally by providing a reverse perspective; we may first use predicates to define behavior as input-output relations and later try to find a concrete behavior that satisfies the Hoare triple. The semantics of SL are as follows.

#### Definition 3.3: Semantics of SL

Let  $\mathcal{V}$  be a finite set of variables. The **semantics** of a program  $S \in \text{SL}$  leads to the following denotational definition of the transition relation  $\llbracket S \rrbracket_{\text{SL}} \subseteq \text{Exec}_{\mathcal{V}}$ , where  $\sigma, \sigma'$  represent the initial and final state, respectively,  $\sigma^{\text{error}}$  is the error state, and  $\zeta : M \rightarrow \text{SL}$  replaces the abstract symbol  $M$  by a program of SL (if defined):

- $\llbracket S; S' \rrbracket_{\text{SL}} = \{(\sigma, \sigma') \mid (\sigma, \sigma_{im}) \in \llbracket S \rrbracket_{\text{SL}}, (\sigma_{im}, \sigma') \in \llbracket S' \rrbracket_{\text{SL}}\}$  with intermediate state  $\sigma_{im}$ ,
- $(\sigma, \sigma') \in \llbracket x := \Theta \rrbracket_{\text{SL}}$  iff  $(\sigma, \sigma') \in \llbracket x := \Theta \rrbracket_{\text{HP}}$ ,
- $(\sigma, \sigma') \in \llbracket \text{havoc } x_1, \dots, x_n \rrbracket_{\text{SL}}$  iff  $(\sigma, \sigma') \in \llbracket x_1 := *; \dots; x_n := * \rrbracket_{\text{HP}}$ ,
- $(\sigma, \sigma') \in \llbracket \text{assume } H \rrbracket_{\text{SL}}$  iff  $(\sigma, \sigma') \in \llbracket ?H \rrbracket_{\text{HP}}$ ,
- $(\sigma, \sigma') \in \llbracket x' = f(x) \ \& \ H \rrbracket_{\text{SL}}$  iff  $(\sigma, \sigma') \in \llbracket x' = f(x) \ \& \ H \rrbracket_{\text{HP}}$ ,
- $(\sigma, \sigma') \in \llbracket \text{if}(H)\{S_1\}\text{else}\{S_2\} \rrbracket_{\text{SL}}$  iff  $(\sigma, \sigma') \in \llbracket \text{assume } H; S_1 \rrbracket_{\text{SL}} \cup \llbracket \text{assume } \neg H; S_2 \rrbracket_{\text{SL}}$ ,
- $\left[ (\sigma, \sigma^{\text{error}}) \in \llbracket \text{assert } H \rrbracket_{\text{SL}} \text{ iff } (\sigma, \sigma) \in \llbracket \text{assume } \neg H \rrbracket_{\text{SL}} \right] \text{ and } \left[ (\sigma, \sigma') \in \llbracket \text{assert } H \rrbracket_{\text{SL}} \text{ iff } (\sigma, \sigma') \in \llbracket \text{assume } H \rrbracket_{\text{SL}} \right]$ .
- $(\sigma, \sigma') \in \llbracket \text{invoke } M \rrbracket_{\text{SL}}$  iff  $\zeta(M)$  exists and  $(\sigma, \sigma') \in \llbracket \zeta(M) \rrbracket_{\text{SL}}$
- $(\sigma, \sigma') \in \llbracket \langle P \rangle M \langle Q \rangle [x_1, \dots, x_n] \rrbracket_{\text{SL}}$  iff  $(\sigma, \sigma') \in \llbracket \text{assert } P; \text{havoc } x_1, \dots, x_n; \text{assume } Q \rrbracket_{\text{SL}}$ .



**Example 3.2.** Consider the thermostat example from [Section 3.1.2](#) and assume we have defined a function  $\xi : M \rightarrow \text{SL}$  that maps an abstract symbol to a program in SL with  $(M_{\text{ON}}, S_{\text{ON}}) \in \xi$ . Assume further that skills **On** and **Heater** are aware of variables  $x, l, u, K, h, s$ , where  $s \in \{1, 2\}$  represents the current state (i.e.,  $1 \mapsto$  heater is activated and  $2 \mapsto$  heater is off). We define the following two behaviors  $S_{\text{ON}}, S_{\text{HEATER}} \in \text{SL}$ :

$$S_{\text{ON}} \equiv \left\{ \begin{array}{l} \text{havoc } h; \text{ assume } h > u; \\ \{x' = -K(h - x) \ \& \ x \leq u\} \end{array} \right\} \quad S_{\text{HEATER}} \equiv \left\{ \begin{array}{l} \text{if } (x \geq u) \{s := 2\} \text{ else } \{\text{skip}\}; \\ \text{if } (x \leq l) \{s := 1\} \text{ else } \{\text{skip}\}; \\ \text{if } (s = 1) \{\text{invoke } M_{\text{ON}}\} \text{ else } \{x' = -Kx \ \& \ l \leq x\} \end{array} \right\}$$

Using the binding in  $\xi$ , we can directly **invoke** program  $S_{\text{ON}}$  within program  $S_{\text{HEATER}}$ , which is semantically equivalent to inlining the program. Alternatively, we may also decide to use construct<sup>4</sup>

$$\text{if } (s = 1) \{ \langle x \leq u \rangle M_{\text{ON}} \langle x \leq u \wedge h > u \wedge \text{past}(x) \leq x \rangle [x, h] \} \text{ else } \{x' = -Kx \ \& \ l \leq x\}$$

in program  $S_{\text{HEATER}}$ , where the behavior of the heater's activated mode is abstracted by a Hoare triple. This abstraction is convenient during program specification and development, as it permits to reason about the correctness of  $S_{\text{HEATER}}$  without developing  $S_{\text{ON}}$  first. Importantly, the addition  $\text{past}(x) \leq x$  is necessary to indicate that temperature  $x$  will only increase. In the refinement process, we may check that  $S_{\text{ON}}$  is indeed a correct implementation of  $\langle x \leq u \rangle M_{\text{ON}} \langle x \leq u \wedge h > u \wedge \text{past}(x) \leq x \rangle [x, h]$  and replace the Hoare triple to get an executable program.

We have almost succeeded in defining a small implementation language to model the behavior of hybrid skills. However, two issues arise in the presence of SL that we briefly discuss.

**Sequential ODEs.** The syntax of SL allows us to sequentially compose systems of ODEs, which is inherited from hybrid programs. Semantically, this means that each system of ODEs runs arbitrarily often (including zero times) before the program continues. This nondeterminism in timing makes sequential compositions of ODEs hard to deal with. Ideally, only a maximum of *one* system of ODEs is executed during the execution of *all* skills, where we additionally define a logical clock together with a worst case execution time. [Example 3.2](#) illustrates this desired behavior for an event-triggered system, where there exist two dissimilar systems of ODEs, but only one of them is executed per run. The goal is to enforce this behavior *per construction*.

**If-Else Cluttering.** The granularity of a skill should still be the responsibility of the developer. However, the more behavior a skill must cover, the more cluttered if-else conditions will be needed. This can already be seen in [Example 3.2](#), where the implementation of  $S_{\text{Heater}}$  is shaped like a transition system, as it essentially covers *two* behaviors: the heater is *on* or *off*. Conversely, the less behavior a skill covers, the more skills must eventually be implemented to represent the desired behavior.<sup>5</sup> This, however, would lead to large skill graphs that lose their visual advantage. Ideally, developers are able to define multiple operating modes for a skill, where each

<sup>4</sup>For convenience, we introduce operator  $\text{past}(\cdot)$  to explicitly evaluate a given variable in its pre-state, as variables in postconditions may evolve based on some ODE.

<sup>5</sup>The described issue is similar to the library scaling problem for components [[Biggerstaff 1994](#)]. On the one hand, too much integrated functionality limits reuse. On the other hand, too specialized components lead to combinatorial growth of components needed.

mode is represented by a program in SL. The goal is to give skills and SL programs more structure, as can be seen in [Section 3.1.2](#), where we used state machines to increase comprehension.

Both issues will be addressed in the next section, where we define *hybrid mode automata* as a structural layer for SL programs.

### 3.2.2. Hybrid Mode Automata

[Example 3.2](#) above illustrates that we view the controlling part of a skill-based maneuver as a (possibly huge) set of operating *modes* of a transition-based system.<sup>6</sup> Furthermore, examples of the *explore world* maneuver consist of the *driving* mode and the *steering* mode. By taking a closer look, the driving mode is *refined* even further: *accelerating*, *cruising*, and *braking* are three possible sub-modes, but are typically not concurrently active. To improve the structuring of SL programs, we introduce a more fine-grained computational model for hybrid skills. Our considerations will lead to a formal notion of *parallel composition* and *mode refinement*, which are both important characteristics from a software engineering perspective.

Parallel composition focuses on the decomposition of the *global behavior* into concurrent and communicating modes. The question which modes can operate simultaneously is answered by an optimistic point of view: two modes can be active at the same time if they do not interfere. Refinement of modes increases the level of abstraction and allows to develop the global behavior step-by-step in a modular fashion. A notion of refinement also directly impacts *reusability* by allowing well-formed combinations of independently developed modes. We develop a formal model of computation and formalize these aspects including the meaning of interference in this section precisely.

### Syntax and Semantics of Hybrid Mode Automata

For modeling computation, we define *hybrid mode automata*, which is a customized combination of *hybrid automata* [[Henzinger 2000](#)] and *mode automata* [[Maraninchi et al. 1998](#); [Maraninchi et al. 2003](#)]. Hybrid mode automata are labeled multidigraphs that consist of a number of discrete modes with guarded transitions between them. An important aspect is that a *discrete* SL program is projected on each mode, which is executed each time a mode is entered. Furthermore, each mode is allowed to have exactly one system of ODEs. The execution semantics of an active mode will follow the semantics of SL (cf. [Definition 3.3](#)). For simplicity, we assume that set  $SL_{disc}$  refers to the domain of possible discrete programs without continuous flow (e.g.,  $\{\text{havoc } a; \text{assume } a \geq 0\}$ ) and that set  $SL_{dyn}$  refers to dynamic systems only (e.g.,  $\{x' = \theta \ \& \ H\}$ ; cf. [Definition 3.2](#)). We then define the syntax of hybrid mode automata as follows.

<sup>6</sup>In this thesis, we distinguish between *modes* and *states* in the context of transition systems to emphasize their level of granularity; compared to elementary states that represent the current valuation of variables, modes may combine a sequence of computations, each leading to a new state.



**Definition 3.4: Hybrid Mode Automata ( $\mathcal{HMA}$ )**

A **hybrid mode automaton**  $\mathcal{A}$  is a tuple  $\langle Q, q^0, V^{\text{in}}, V^{\text{out}}, \text{Trans}, \text{Dyn}, \text{Ctrl} \rangle$  where:

- $Q$  is a finite set of modes,
- $q^0 \in Q$  is the initial mode,
- $V^{\text{in}}$  and  $V^{\text{out}}$  are sets of input and output variables in  $\mathbb{R}$ , respectively. We require that  $V^{\text{in}} \cap V^{\text{out}} = \emptyset$ ,
- $\text{Trans} \subseteq Q \times G(V) \times Q$  is the set of transitions, which are labeled by a first-order logical formula  $G$  over the input and output variables. We use the notation  $q \xrightarrow{g} q'$  for  $(q, g, q') \in \text{Trans}$ ,
- $\text{Dyn} : Q \rightarrow \text{SL}_{\text{dyn}}$  maps a mode to a (possibly empty) system of ordinary differential equations,
- $\text{Ctrl} : Q \rightarrow \text{SL}_{\text{disc}}$  maps a mode to the discrete control part.

We denote the **universe** of all hybrid mode automata by  $\mathcal{HMA}$ .

Each mode is associated with a sequence of *discrete computations* based on  $\text{SL}_{\text{disc}}$ . Again, we excluded *non-deterministic choice* and *repetition*. Arbitrary non-deterministic choice allows to model families of controllers, but needs to be resolved when deriving concrete implementations. For example, a safety goal may be ensured whether we decide to accelerate or brake, but an operating controller should not alternate between these two options every other cycle. We bypass this complication by focusing on controllers that are already close to the intended behavior. However, to retain some flexibility, we still allow *nondeterministic assignment*. All abstractions, including nondeterministic assignment and Hoare triples, need to be resolved when deriving a concrete implementation and it needs to be ensured (i.e., proven correct) that the resolution preserves the modeled behavior.

In addition to **Definition 3.4**, an  $\mathcal{A} \in \mathcal{HMA}$  must satisfy the following correctness conditions:

- C<sub>1</sub>:**  $\mathcal{A}$  is deterministic in the sense that for each mode  $q \in Q$ , if there exist outgoing transitions  $(q, g_1, q'), (q, g_2, q'')$  with  $q' \neq q''$ , then  $g_1 \wedge g_2$  is not satisfiable.
- C<sub>2</sub>:** After each *cycle*,  $\mathcal{A}$  must take a transition. That is,  $\forall q \in Q : \bigvee_{(q, g, q') \in \text{Trans}} g$  is true. If no transition can be taken, we assume an implicit self-transition  $(q, g_q, q)$  exists with  $g_q \equiv \neg(\bigvee_{(q, g, q') \in \text{Trans}} g)$  ( $g_q \equiv \text{true}$  if no outgoing transitions exist). If an explicit self-transition exists but no transition can be taken, then  $\mathcal{A}$  is ill-formed (i.e., not a hybrid mode automaton).

Furthermore, we introduce a third and fourth unique category of variables, namely *global variables* (denoted  $\mathcal{V}^{\text{global}}$ ) and *local variables* (denoted  $\mathcal{V}^{\text{local}}$ ). Global variables can be globally accessed by any hybrid mode automaton and are (for simplicity) added to each automaton as part of the input variables. Global variables are read-only and assumed to be initialized only once in the beginning by the global system. Typical examples are *constants*, such as the upper ( $u$ ) and lower ( $l$ ) limit of the

thermostat system. Local variables are introduced as part of the discrete program and only visible to the automaton itself.<sup>7</sup> Therefore, the following correctness condition must also hold:

**C<sub>3</sub>:** Output variables of any hybrid mode automaton are disjoint from the global variables (i.e.,  $V^{\text{out}} \cap V^{\text{global}} = \emptyset$ ) and local variables are disjoint from input and output variables (i.e.,  $(V^{\text{in}} \cup V^{\text{out}}) \cap V^{\text{local}} = \emptyset$ ).

**Example 3.3.** Consider again the thermostat system. If we ignore the partition into skills for now, then an event-triggered representation (i.e., without considering timing) of the overall behavior as a hybrid mode automaton  $\langle Q, q^0, V^{\text{in}}, V^{\text{out}}, \text{Trans}, \text{Dyn}, \text{Ctrl} \rangle$  is given as follows:

$$\begin{array}{ll}
 Q &= \{ \text{On}, \text{Off} \} \\
 q^0 &= \{ \text{On} \} \\
 V^{\text{in}} &= \{ x \} \cup V^{\text{global}} \\
 V^{\text{out}} &= \{ h \} \\
 V^{\text{global}} &= \{ u, l, K \}
 \end{array}
 \quad
 \begin{array}{ll}
 \text{Trans} &= \{ (\text{On}, x \geq u, \text{Off}), (\text{Off}, x \leq l, \text{On}) \} \\
 \text{Dyn}(q) &= \begin{cases} \{ x' = K(h - x) \ \& \ x \leq u \} & \text{if } q = \text{On} \\ \{ x' = -Kx \ \& \ x \geq l \} & \text{if } q = \text{Off} \end{cases} \\
 \text{Ctrl}(q) &= \begin{cases} \{ \text{havoc } h; \text{assume } h > u \} & \text{if } q = \text{On} \\ \{ \text{assume true} \} & \text{if } q = \text{Off} \end{cases}
 \end{array}$$

The controller of mode [On] nondeterministically chooses a heating power with condition  $h > u$ , whereas the controller of mode [Off] does nothing. The evolution constraint of both dynamic systems makes sure that temperature  $x$  stays between lower limit  $l$  and upper limit  $u$ . Later, we will see how timing can be incorporated to make the shift to more realistic time-triggered systems.

The semantics of  $\mathcal{HMA}$  is given in terms of *valid execution sequences* (or traces, respectively). Informally, an execution trace of a hybrid mode automaton is a sequence of modes together with input and output state. The input state is the evaluation of all variables before executing a mode and the output state is the result. In each control cycle, the automaton transits to a mode (possibly to itself) and instantaneously updates all variables in the current state according to the attached discrete program. Afterwards, the continuous evolution according to the attached dynamic system progresses for an arbitrary amount of time (see [Definition 2.5](#)). A convenience of our following formalization will be that free variables are only associated with the *previous* state. For clarification, consider the discrete program  $[x := x + 1; x := x + 1]$ , where  $x$  is incremented twice. In the following semantics, we view this program as  $[x^- := x; x := x^- + 1; x := x^- + 1]$ , where  $x^-$  refers to  $x$  as evaluated in the input state. Consequently, the two programs  $[x := x + 1; x := x + 1]$  and  $[x := x + 1]$  are equivalent according to our semantics, and  $x$  is only incremented by 1 in both cases. This way, we have a clear semantics for difficult constructs, such as cycles (i.e.,  $[x := y + 1; y := x + 1]$ ), and offer a synchronization point for *parallel composition* (more on that later). We give a formal definition of the semantics of hybrid mode automata in [Definition 3.5](#).

<sup>7</sup>In the remainder of this chapter, we mostly ignore local variables as part of our formalization. We assume that (1) they are always disjoint from input and output variables, and (2) in case of name clashes, adequate renaming and substitution suffices.

**Definition 3.5: Execution Semantics of Hybrid Mode Automata**

Let  $\sigma_i^{\text{in}}$  and  $\sigma_i^{\text{out}}$  denote valuations of variables in  $V$ . A **valid run** of a hybrid mode automaton  $\mathcal{A} = \langle Q, q^0, V^{\text{in}}, V^{\text{out}}, \text{Trans}, \text{Dyn}, \text{Ctrl} \rangle$  is a sequence of mode switches  $l_0, \dots, l_n \in Q$  of the following form

$$\text{run}_{\mathcal{A}} = \langle \sigma_0^{\text{in}}, l_0, \sigma_0^{\text{out}} \rangle, \dots, \langle \sigma_n^{\text{in}}, l_n, \sigma_n^{\text{out}} \rangle$$

such that

- $\sigma_i^{\text{in}}$  and  $\sigma_i^{\text{out}}$  are input and output valuations of variables in  $V$  in mode  $l_i$ ,
- for all  $k = 0, \dots, n$ ,

$$(\sigma_k^{\text{in}}, \sigma_k^{\text{out}}) \in \llbracket \text{Mem}(l_k); \text{Ctrl}(l_k)[v_i \mapsto v_i^-]_{\text{rhs}}^V; \text{Dyn}(l_k)[v_i \mapsto v_i^-]_{\text{rhs}}^V \rrbracket_{\text{SL}}$$

with

$$\text{Mem}(l_k) = v_1^- := v_1; \dots; v_n^- := v_n \quad \forall v_i \in V.$$

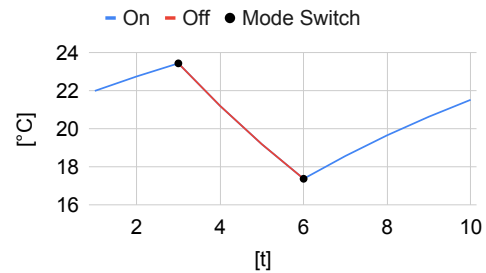
and  $[v_i \mapsto v_i^-]_{\text{rhs}}^V$  is an operator that replaces all occurrences of variables  $v_i \in V$  on the right-hand side of expressions or ODEs with the fresh variable  $v_i^-$ ,

- the initial execution is performed after taking the first transition from initial mode  $q^0$ , such that  $\langle q^0, G, l_0 \rangle \in \text{Trans}$  and  $\sigma_0^{\text{in}} \models G$ ,
- for each  $i = 0, \dots, n-1$ ,  $\langle \sigma_i^{\text{in}}, l_i, \sigma_i^{\text{out}} \rangle$  is followed by  $\langle \sigma_{i+1}^{\text{in}}, l_{i+1}, \sigma_{i+1}^{\text{out}} \rangle$  if and only if there exists a transition  $(l_i, G, l_{i+1}) \in \text{Trans}$  and  $\sigma_{i+1}^{\text{in}} \models G$ .

The length of a run corresponds with the cycles considered. Per definition, each prefix of a valid run  $\text{run}_{\mathcal{A}}$  of a hybrid mode automaton  $\mathcal{A}$  (including the empty run  $\text{run}_{\mathcal{A}} = \epsilon$ ) is therefore also a valid run of  $\mathcal{A}$ .

**Example 3.4.** Consider the thermostat example in [Figure 3.4](#) with modes [On] and [Off], and assume that the following concrete values are used: initial  $x = 22^\circ\text{C}$ , upper limit  $u = 24^\circ\text{C}$ , lower limit  $l = 16^\circ\text{C}$ ,  $\delta_l = 2^\circ\text{C}$ ,  $\delta_u = 1^\circ\text{C}$ , room temperature constant  $K = 0.1$ , and constant heating power  $h = 30^\circ\text{C}$ . That is, the heater is switched to [Off] if the temperature rises above  $23^\circ\text{C}$  and switched to [On], if the temperature falls below  $18^\circ\text{C}$ . With initial model [On], one of many valid runs has the form:

$$\begin{aligned} & \langle \{x = 22\}, \text{On}, \{x = 22.76\} \rangle_{t=1} \\ \longrightarrow & \langle \{x = 22.76\}, \text{On}, \{x = 23.45\} \rangle_{t=2} \\ \longrightarrow & \langle \{x = 23.45\}, \text{Off}, \{x = 21.21\} \rangle_{t=3} \\ \longrightarrow & \dots \end{aligned}$$



In this scenario, we used a step size of 1 to evaluate whether a mode switch is necessary, which is also why the discrepancy in input and output temperature appears so high. In this case, a step size of 1 is enough to keep the temperature between  $16^\circ\text{C}$  and  $24^\circ\text{C}$ .

In the next sections, we use hybrid mode automata to formally describe the behavior of hybrid skills. Before that, we will discuss two important operations as part of our formalism, namely *parallel*

*composition* and *mode refinement*.<sup>8</sup> For any larger automaton, it is reasonable to imagine that we may start with smaller automata and subsequently combine them to eventually form the larger one. Parallel composition as an operator takes two hybrid mode automata and combines them to a third one that expresses the behaviors of the first two simultaneously. Refinement as an operator substitutes a single mode with a complete hybrid mode automaton. Both operations are useful in the context of skill graphs, as they allow us to develop hybrid mode automata in isolation and subsequently combine them automatically. This is in alignment with skills in a skill graph that must also be combined vertically and horizontally. Of course, both operations are governed by constraints that we will discuss alongside the formalization in the following.

## Parallel Composition

As mentioned before, *parallel composition* of two hybrid mode automata  $\mathcal{A}_1$  and  $\mathcal{A}_2$  results in a new hybrid mode automaton  $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$  that represents the simultaneous execution of both automata.  $\mathcal{A}_1$  and  $\mathcal{A}_2$  do not need to be completely decoupled (i.e., without any information flow between them), but it only makes sense to define compositions of two hybrid mode automata that are *compatible*. For instance, consider again the *Explore World* maneuver of the vacuum cleaner of the previous section. Although skill **Accelerate** and skill **Yaw** are never part of the same *mode*, we may also think about activating both behaviors concurrently to let the robot drive curves. In contrast, accelerating and decelerating at the same time does not seem to be useful, which also leads to confusion on a technical level; both skills will (theoretically) write on variable  $a$  simultaneously.

Before we define, how parallel composition (or simply composition) of hybrid mode automata looks like, we first define the (valid) composition of dynamic systems. Informally, the composition of dynamic systems is simply their combined continuous behavior. Validity is therefore based on *agreement*; first, their evolution domains must not be in conflict and second, they are not allowed to let the same variable evolve *differently*.

### Definition 3.6: Composition of Dynamic Systems

Let  $d_1 = \{X'_1 = \Theta_1 \& H_1\}$ ,  $d_2 = \{X'_2 = \Theta_2 \& H_2\} \in \text{SL}_{\text{dyn}}$  be two dynamic systems. The **composition** of  $d_1$  and  $d_2$  is defined by the operation  $\oplus : \text{SL}_{\text{dyn}} \times \text{SL}_{\text{dyn}} \rightarrow \text{SL}_{\text{dyn}}$ :

$$d_1 \oplus d_2 = \{X'_1 = \Theta_1, X'_2 = \Theta_2 \& H_1 \wedge H_2\}.$$

We consider a composition of dynamic systems to be **valid** iff the following conditions hold:

- Evolution domain  $H_1$  does not conflict with evolution domain  $H_2$  (i.e.,  $H_1 \wedge H_2$  is satisfiable),
- ODEs of shared bound variables have to be identical (i.e.,  $\forall (x'_1 = \theta_1) \in d_1, (x'_2 = \theta_2) \in d_2 : x'_1 = x'_2 \Rightarrow \theta_1 = \theta_2$ ).

<sup>8</sup>In the literature, this kind of refinement is also sometimes referred to as *vertical* or *hierarchical composition*, whereas parallel composition is sometimes referred to as *horizontal composition*.

Based on the previous considerations, our condition for *composability* is twofold. First, the sets of output variables of both hybrid mode automata must be disjoint. Second, the dynamical systems should not conflict with each other.

**Definition 3.7: Composable Hybrid Mode Automata**

Two hybrid mode automata  $\mathcal{A}_1$  and  $\mathcal{A}_2$  are **composable** iff (1)  $V_1^{\text{out}} \cap V_2^{\text{out}} = \emptyset$  and (2)  $\forall q_1 \in Q_1, q_2 \in Q_2 : \text{Dyn}_1(q_1) \oplus \text{Dyn}_2(q_2)$  is a valid composition of dynamical systems.

Parallel composition is constructed as usual. In principle, this operation represents the cartesian product of two automata, where each new mode inherits the behaviors and dynamics of the original pair of modes. Additionally, the output variables of both original automata will still be visible, whereas input variables will only be visible if one of the automata does not declare it as an output variable.

**Definition 3.8: Parallel Composition of Hybrid Mode Automata**

Let  $\mathcal{A}_1, \mathcal{A}_2 \in \mathcal{HMA}$  be two composable hybrid mode automata. We define the **parallel composition** by

$$\mathcal{A}_1 \parallel \mathcal{A}_2 = \langle Q_1 \times Q_2, (q_1^0, q_2^0), V_1^{\text{in}} \cup V_2^{\text{in}} \setminus (V_1^{\text{out}} \cup V_2^{\text{out}}), V_1^{\text{out}} \cup V_2^{\text{out}}, \text{Trans}, \text{Dyn}, \text{Ctrl} \rangle$$

where:

- $((q_1, q_2), G, (q'_1, q'_2)) \in \text{Trans}$  iff  $(q_1, G_1, q'_1) \in \text{Trans}_1$  and  $(q_2, G_2, q'_2) \in \text{Trans}_2$  with  $G = G_1 \wedge G_2$ ,
- $\forall (q_1, q_2) \in Q_1 \times Q_2 : \text{Dyn}(q_1, q_2) = \text{Dyn}_1(q_1) \oplus \text{Dyn}_2(q_2)$ ,
- $\forall (q_1, q_2) \in Q_1 \times Q_2 : \text{Ctrl}(q_1, q_2) = \text{Ctrl}_1(q_1) ; \text{Ctrl}_2(q_2)$ .

Parallel composition of two hybrid mode automata is straightforward. The only interesting parts are the dynamic system and the discrete controller per mode. For the dynamic system, we simply build the union of the individual dynamic systems and conjoin the evolution constraint. For the discrete controller, we use *sequential composition* for the individual discrete controllers. We provide the following **Theorem 3.2** to state that the class of  $\mathcal{HMA}$  is closed under composition. That is, larger  $\mathcal{HMA}$  can be build from simpler ones in an inductive fashion by only checking *composability*.

**Theorem 3.1: Closed under Composition**

Let  $\mathcal{A}_1, \mathcal{A}_2 \in \mathcal{HMA}$  be two composable hybrid mode automata. Then, their composition  $\mathcal{A}_1 \parallel \mathcal{A}_2$  yields also a hybrid mode automaton.

**Proof.** We need to show that  $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$  satisfies the correctness conditions  $C_1$ ,  $C_2$ , and  $C_3$ .

**$C_1$ .** Assume there exist  $((q_1, q_2), G_1, (q'_1, q'_2)), ((q_1, q_2), G_2, (q''_1, q''_2)) \in \text{Trans}$  and  $G_1 \wedge G_2$  is satisfiable. W.l.o.g., assume further that  $G_1 \equiv g_1 \wedge g_3$  and  $G_2 \equiv g_2 \wedge g_4$  are the

respective conjoined guards from the parallel composition with the initial transitions  $(q_1, g_1, q'_1), (q_1, g_2, q''_1) \in \text{Trans}_1$  and  $(q_2, g_3, q'_2), (q_2, g_4, q''_2) \in \text{Trans}_2$ . If  $G_1 \wedge G_2 = (g_1 \wedge g_3) \wedge (g_2 \wedge g_4)$  is satisfiable, then so are  $g_1 \wedge g_2$  and  $g_3 \wedge g_4$ , which is a contradiction to  $\mathcal{A}_1, \mathcal{A}_2 \in \mathcal{HMA}$ . Hence,  $G_1$  or  $G_2$  cannot be satisfiable at the same time.

- C<sub>2</sub>. If for a fixed  $q_1 \in Q_1$  and a fixed  $q_2 \in Q_2$  the formulas  $\bigvee_{(q_1, g_1^i, q'_1) \in \text{Trans}_1} g_1^i$  and  $\bigvee_{(q_2, g_2^j, q'_2) \in \text{Trans}_2} g_2^j$  for  $i = 1, \dots, n$  and  $j = 1, \dots, m$  both evaluate to true, so does  $\bigvee_{((q_1, q_2), g_1^i \wedge g_2^j, (q'_1, q'_2)) \in \text{Trans}} g_1^i \wedge g_2^j$  for a fixed  $(q_1, q_2) \in Q$ , as:

$$(g_1^1 \vee \dots \vee g_1^n) \wedge (g_2^1 \vee \dots \vee g_2^m) \equiv (g_1^1 \wedge g_2^1) \vee (g_1^1 \wedge g_2^2) \vee \dots \vee (g_1^n \wedge g_2^m).$$

- C<sub>3</sub>. Follows directly from the definition of parallel composition, which does not add global variables. Local variables are ignored in [Definition 3.8](#), but are essentially unified. As mentioned before, conflicts can easily be resolved by renaming.

□

After having succeeded in constructing a valid composition of hybrid mode automata, we now establish two important properties of our definition, namely *commutativity* and *associativity*. Commutativity allows us to *ignore* the order when composing two automata, as the resulting automaton will always be semantically equivalent. Associativity ensures that our *binary* composition operator is enough, even for an arbitrary amount of composable automata, as it allows us to build larger automata incrementally. To state and proof both properties concisely, we first define the meaning of non-interference between two discrete programs in SL and an additional lemma afterwards.

#### Definition 3.9: Non-Interference of Discrete SL Programs

Two programs  $S_1, S_2 \in \text{SL}_{\text{disc}}$  are **non-interfering** iff  $\text{bound}(S_1) \cap \text{var}(S_2) = \emptyset$  and  $\text{bound}(S_2) \cap \text{var}(S_1) = \emptyset$ .

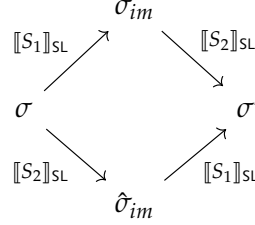
Non-interference means that both programs do not write on the same variables. We ignore the dynamical part, as this is not important for our upcoming considerations.

#### Lemma 3.1: Order-Irrelevance of Sequential Composition

If two programs  $S_1, S_2 \in \text{SL}_{\text{disc}}$  are non-interfering, then  $\llbracket S_1 ; S_2 \rrbracket_{\text{SL}} = \llbracket S_2 ; S_1 \rrbracket_{\text{SL}}$ .

**Proof.** Follows from the semantics of SL programs (see [Definition 3.3](#)). Assume  $(\sigma, \sigma_{im}) \in \llbracket S_1 \rrbracket_{\text{SL}}$  and  $(\sigma_{im}, \sigma') \in \llbracket S_2 \rrbracket_{\text{SL}}$ . We need to show that there exists  $(\sigma, \hat{\sigma}_{im}) \in \llbracket S_2 \rrbracket_{\text{SL}}$  and  $(\hat{\sigma}_{im}, \sigma') \in \llbracket S_1 \rrbracket_{\text{SL}}$ . As both programs  $S_1$  and  $S_2$  are non-interfering, their effects on bound variables is separable. That is, only bound variables in  $S_1$  are affected by  $\llbracket S_2 \rrbracket_{\text{SL}}$  and analogously for  $S_2$  (i.e., on the bound variables in  $S_1$ ,  $\sigma_{im}(x) = \sigma(x)$  for  $S_2$ ). It follows that  $\sigma_{im} = \bigcup_{x \in \text{bound}(S_1)} \sigma'(x) \cup \bigcup_{x \in \text{bound}(S_2)} \sigma(x)$  is a valid intermediate state for  $\llbracket S_1 ; S_2 \rrbracket$ . Applying the same reasoning,  $\hat{\sigma}_{im} = \bigcup_{x \in \text{bound}(S_2)} \sigma'(x) \cup$

$\bigcup_{x \in \text{bound}(S_1)} \sigma(x)$  is likewise a valid intermediate state for  $\llbracket S_2; S_1 \rrbracket$ . The commuting diagram visualizes this reasoning:



□

### Proposition 3.1: Algebraic Properties of Parallel Composition: Commutativity

The parallel composition of composable hybrid mode automata is *commutative*.

**Proof.** Let  $\mathcal{A}_1, \mathcal{A}_2 \in \mathcal{HMA}$  be two composable hybrid mode automata. We show that  $\mathcal{A}_1 \parallel \mathcal{A}_2$  is semantically equivalent to  $\mathcal{A}_2 \parallel \mathcal{A}_1$ .

Let  $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2 = \langle Q, q^0, V^{\text{in}}, V^{\text{out}}, \text{Trans}, \text{Dyn}, \text{Ctrl} \rangle$ . A semantically equivalent hybrid mode automata  $\mathcal{A}'$  is obtained by re-defining  $Q' = \{(q_2, q_1) \mid (q_1, q_2) \in Q\}$ ,  $q'^0 = (q_2, q_1)$ , and  $\text{Trans}' = \{((q_2, q_1), G, (q'_2, q'_1)) \mid ((q_1, q_2), G, (q'_1, q'_2)) \in \text{Trans}\}$ , while the other elements remain equal for now. The critical part is the projected program of a mode  $q \in Q$  (i.e.,  $\text{Ctrl}(q)$ ), which results from the sequential composition of  $\text{Ctrl}_1(q_1)$  and  $\text{Ctrl}_2(q_2)$ . Based on semantics of hybrid mode automata (Definition 3.5), all free variables of the projected program are replaced by fresh variables beforehand, which we denote by  $\overline{\text{Ctrl}}_i$ . It follows that

$$\begin{aligned} \text{bound}(\overline{\text{Ctrl}}_1(q_1)) \cap \text{free}(\overline{\text{Ctrl}}_2(q_2)) &= \emptyset \\ \wedge \text{bound}(\overline{\text{Ctrl}}_2(q_2)) \cap \text{free}(\overline{\text{Ctrl}}_1(q_1)) &= \emptyset. \end{aligned}$$

As  $\mathcal{A}_1$  and  $\mathcal{A}_2$  are composable, it follows that  $\overline{\text{Ctrl}}_1(q_1)$  and  $\overline{\text{Ctrl}}_2(q_2)$  are non-interfering, and by applying Lemma 3.1, we obtain  $\llbracket \overline{\text{Ctrl}}_1(q_1); \overline{\text{Ctrl}}_2(q_2) \rrbracket_{\text{SL}} = \llbracket \overline{\text{Ctrl}}_2(q_2); \overline{\text{Ctrl}}_1(q_1) \rrbracket_{\text{SL}}$ . Composition of the dynamic system is valid, as  $\mathcal{A}_1$  and  $\mathcal{A}_2$  are composable, and commutative per Definition 3.6 (i.e., conjunction of evolution constraint is commutative). Finally, this process leads to the hybrid mode automata  $\mathcal{A}' = \mathcal{A}_2 \parallel \mathcal{A}_1$ , which is semantically equivalent to  $\mathcal{A}$ .

□

### Proposition 3.2: Algebraic Properties of Parallel Composition: Associativity

The parallel composition of composable hybrid mode automata is *associative*.

**Proof.** Let  $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3 \in \mathcal{HMA}$  be three hybrid mode automata that are pairwise composable. We show that  $(\mathcal{A}_1 \parallel \mathcal{A}_2) \parallel \mathcal{A}_3$  is semantically equivalent to  $\mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3)$ . We identify the following equivalences:



- Cartesian product is associative up to isomorphism:  $(Q_1 \times Q_2) \times Q_3 = Q_1 \times (Q_2 \times Q_3)$  and  $((q_1^0, q_2^0), q_3^0) = (q_1^0, (q_2^0, q_3^0))$ .
- $(V_{1,2}^{\text{in}} \cup V_3^{\text{in}}) \setminus ((V_1^{\text{out}} \cup V_2^{\text{out}}) \cup V_3^{\text{out}})$  is equivalent to  $(V_1^{\text{in}} \cup V_2^{\text{in}} \cup V_3^{\text{in}}) \setminus (V_1^{\text{out}} \cup V_2^{\text{out}} \cup V_3^{\text{out}})$  with  $V_{1,2}^{\text{in}} = V_1^{\text{in}} \cup V_2^{\text{in}} \setminus (V_1^{\text{out}} \cup V_2^{\text{out}})$ . Analogously for  $(V_1^{\text{in}} \cup V_{2,3}^{\text{in}}) \setminus (V_1^{\text{out}} \cup (V_2^{\text{out}} \cup V_3^{\text{out}}))$ .
- Union is associative:  $(V_1^{\text{out}} \cup V_2^{\text{out}}) \cup V_3^{\text{out}} = V_1^{\text{out}} \cup (V_2^{\text{out}} \cup V_3^{\text{out}})$
- $((q_1, q_2), q_3), (G_1 \wedge G_2) \wedge G_3, ((q'_1, q'_2), q'_3)) \in \text{Trans}_{(1,2),3} = ((q_1, (q_2, q_3)), G_1 \wedge (G_2 \wedge G_3), (q'_1, (q'_2, q'_3))) \in \text{Trans}_{1,(2,3)} = ((q_1, q_2, q_3), G_1 \wedge G_2 \wedge G_3, (q'_1, q'_2, q'_3)) \in \text{Trans}_{1,2,3}$
- Finally, sequential composition of SL programs and composition of differential equations are both associative (see [Definition 3.3](#)).

□

## Mode Refinement

The second operation we introduce is *refinement* of modes inspired by Maraninchi et al. [2003]. As mentioned before, this operation takes a mode  $q_1 \in Q_1$  in  $\mathcal{A}_1$  and a hybrid mode automaton  $\mathcal{A}_2$  as input and produces a new hybrid mode automaton  $\mathcal{A}$ .  $\mathcal{A}$  is almost identical to  $\mathcal{A}_1$ , but composes mode  $q_1$  with automaton  $\mathcal{A}_2$  in a particular way. This operator is especially interesting, as it allows to develop hybrid mode automata *incrementally*. The idea is to start with an abstract vision of a mode and fill in the necessary details later when needed, either from scratch or automatically by reusing already constructed hybrid mode automata. This is in accordance with our *abstraction* constructs in SL, namely *invocations* and *Hoare triples*. Mode refinement allows us to either *inline* a complete automaton or to replace a Hoare triple with it. Similar to parallel composition, refinements *add* detail to already existing modes (e.g., with possibly non-empty discrete and dynamic control parts), but should somehow adhere to the purpose of the original mode. This means that a refinement of a mode is regulated by a set of rules and cannot arbitrarily change the mode's intent in behavior and dynamics.

One example of such a regulation is the dynamic system. In a flattened version of a hybrid mode automaton, each mode is associated with a (possibly empty) set of differential equations. Elements of that set should not contradict each other (e.g., having ODEs with dissimilar solutions for the same variable). Parallel composition bypasses this problem per definition (see [Definition 3.7](#)). Mode refinement does not enjoy this property. The question is therefore how a refinement of dynamical systems may look like. Conveniently, the definition of the composition of dynamic systems (see [Definition 3.6](#)) already resolves this issue on its own by requiring that each bound variable corresponds to exactly one differential equation. It is therefore convenient to reuse the composition of dynamic systems *almost* as-is and require that mode refinement is only possible, if the composition of dynamical systems is valid. We adapt the composition operator for refinement slightly and require for the composition of two dynamical systems  $\{X'_1 = \Theta_1 \ \& \ H_1\} \oplus \{X'_2 = \Theta_2 \ \& \ H_2\}$  that  $H_2 \Rightarrow H_1$  must be satisfiable (instead of  $H_1 \wedge H_2$ ). This means that mode refinement cannot *increase* the valid region given by evolution constraints, but must operate in the existing one. We will use the notation  $\oplus$  to indicate this difference.



Relying on the composition of dynamic systems for mode refinement is somewhat coarse-grained, but easy to implement and apply. For instance, we prohibit refinement and overwriting of differential equations. Instead, we require that differential equations for the same variable are identical if they are part of both dynamic systems.<sup>9</sup> We also do not make the implicit assumption that a non-existent ODE for an output variable  $y$  is synonymous to  $(y' = 0)$ , as this would violate a valid refinement by definition. Instead, such ODEs must be provided explicitly to restrict possible refinements.

#### Definition 3.10: Refinement of Hybrid Mode Automata

Let  $\mathcal{A}_1 \in \mathcal{HMA}$  be a hybrid mode automaton and  $q_1 \in Q_1$  any of its modes. A **refinement** of mode  $q_1$  by  $\mathcal{A}_2 \in \mathcal{HMA}$  is defined by the operation  $\blacktriangleright: Q_1 \times \mathcal{HMA} \rightarrow \mathcal{HMA}$ :

$$q_1 \blacktriangleright \mathcal{A}_2 = \langle Q, q^0, (V_1^{\text{in}} \cup V_2^{\text{in}}) \setminus (V_1^{\text{out}} \cup V_2^{\text{out}}), V_1^{\text{out}} \cup V_2^{\text{out}}, \text{Trans}, \text{Dyn}, \text{Ctrl} \rangle$$

where

- $Q = Q_1 \setminus \{q_1\} \cup Q_2.$
- $q^0 = \begin{cases} q_1^0: q_1^0 \neq q_1 \\ q_2^0: \text{otherwise} \end{cases}$
- $\text{Trans} = \{(q, g, q') \in \text{Trans}_1 \mid q \neq q_1 \wedge q' \neq q_1\}$  (keep valid transitions of  $\mathcal{A}_1$ )  
 $\cup \{(q, g, q_2^0) \mid (q, g, q_1) \in \text{Trans}_1 \wedge q \neq q_1\}$  (handle incoming transitions of  $q_1$ )  
 $\cup \{(q_2, g, q) \mid q_2 \in Q_2 \wedge (q_1, g, q) \in \text{Trans}_1 \wedge q_1 \neq q\}$  (handle outgoing transitions of  $q_1$ )  
 $\cup \{(q, g \wedge g_{q_1}, q') \mid (q, g, q') \in \text{Trans}_2, (q_1, g_{q_1}, q_1) \in \text{Trans}_1\}$  (add transitions of  $\mathcal{A}_2$ )
- $\text{Dyn}(q) = \begin{cases} \text{Dyn}_1(q) & : q \in Q_1 \\ \text{Dyn}_1(q_1) \oplus \text{Dyn}_2(q) & : q \in Q_2 \end{cases}$
- $\text{Ctrl}(q) = \begin{cases} \text{Ctrl}_1(q) & : q \in Q_1 \\ \text{Ctrl}_1(q_1); \text{Ctrl}_2(q) & : q \in Q_2 \end{cases}$

**Example 3.5.** In the thermostat example, we describe behaviors of skill **Heater** and **On** as follows using hybrid mode automata  $\mathcal{A}_{\text{Heater}}$  (on the left) and  $\mathcal{A}_{\text{On}}$  (on the right):

$Q$	$= \{AOn, Off\}$	$Q'$	$= \{On\}$
$q^0$	$= \{AOn\}$	$q'^0$	$= \{On\}$
$V^{\text{in}}$	$= \{x\} \cup V^{\text{global}}$	$V'^{\text{in}}$	$= \{x\} \cup V^{\text{global}}$
$V^{\text{out}}$	$= \{h\}$	$V'^{\text{out}}$	$= \{h\}$
$V^{\text{global}}$	$= \{u, l, K\}$	$V'^{\text{global}}$	$= \{u, K, l\}$
$\text{Trans}$	$= \{(AOn, x \geq u, Off), (Off, x \leq l, AOn)\}$	$\text{Trans}'$	$= \{\}$
$\text{Dyn}(q)$	$= \begin{cases} \epsilon & \text{if } q = AOn \\ \{x' = -Kx \ \& \ x \geq l\} & \text{if } q = Off \end{cases}$	$\text{Dyn}'(q)$	$= \{x' = K(h - x) \ \& \ x \leq u\}$
$\text{Ctrl}(q)$	$= \begin{cases} \{\text{invoke } On\} & \text{if } q = AOn \\ \{\text{assume } true\} & \text{if } q = Off \end{cases}$	$\text{Ctrl}'(q)$	$= \{\text{havoc } h; \text{assume } h > u\}$

<sup>9</sup>For a more fine-grained solution, Loos et al. [2016] proposed *differential refinement logic* (dRCL), which elaborates on refinement of ODEs based on reachable states. However, this is only possible for event-based systems.

Let  $\downarrow: \mathcal{HMA} \times Q \rightarrow Q$  denote the projection of a hybrid mode automaton onto one of its modes. Mode refinement  $(\mathcal{A}_{Heater} \downarrow \mathcal{A}_{On}) \blacktriangleright \mathcal{A}_{On}$  yields the hybrid automaton presented in [Example 3.3](#), if we remove *invoke On* after mode refinement (which is reasonable to do).

Similar to parallel composition and composability of hybrid mode automata, only specific refinements should be allowed. We define the class of admissible refinements as follows.

#### Definition 3.11: Admissible Mode Refinement

Let  $\mathcal{A}_1, \mathcal{A}_2 \in \mathcal{HMA}$  be two hybrid mode automata and  $q_1 \in Q_1$  a mode of  $\mathcal{A}_1$ . An **admissible mode refinement**  $q_1 \blacktriangleright \mathcal{A}_2 \in \mathcal{HMA}$  satisfies the following correctness constraints:

- for all  $q \in Q_2$ ,  $\text{Dyn}_1(q_1) \oplus \text{Dyn}_2(q)$  is a *valid* composition of dynamic systems,
- for all  $q \in Q_2$ , the discrete programs of  $q_1$  and  $q$  are not writing on the same output variables (i.e.,  $\text{bound}(\text{Ctrl}_1(q_1)) \cap \text{bound}(\text{Ctrl}_2(q)) = \emptyset$ ),

#### Theorem 3.2: Closed under Mode Refinement

Let  $\mathcal{A}_1, \mathcal{A}_2 \in \mathcal{HMA}$  be two hybrid mode automata,  $q_1 \in Q_1$  a mode of  $\mathcal{A}_1$ , and refinement  $q_1 \blacktriangleright \mathcal{A}_2 \in \mathcal{HMA}$  be admissible. Then,  $q_1 \blacktriangleright \mathcal{A}_2 \in \mathcal{HMA}$  yields also a hybrid mode automaton.

**Proof.** For brevity, we only present the proof idea here, as this is similar to parallel composition. Observe that mode refinement is essentially composition of *one* mode with an automaton instead of building the cartesian product. The definition of admissible mode refinement (see [Definition 3.11](#)) is almost identical to the definition of composability (see [Definition 3.7](#)), and indeed even implied. Likewise, guards of the new transitions from  $\mathcal{A}_2$  are conjoined with the guard of  $q_1$ 's self-transition. It can be shown that if the class of  $\mathcal{HMA}$  is closed under composition, then it must certainly be closed under mode refinement.  $\square$

## Extended Syntax for Composition and Mode Refinement

A hybrid mode automaton allows us now to *organize* a finite number of modes, which, however, are not all always relevant in particular maneuvers. If we consider again the *explore world* maneuver from [Section 3.1.2](#), we see that skill **Control Dynamics** provides multiple actions depending on an input command  $c \in \{1, \dots, 3\}$ ; *acceleration and cruising*, *braking*, and *turning*. Besides the initiation mode, only acceleration and cruising have a transition relationship, whereas the other two must be directly activated by input  $c$ . The top-level behavior controls this activation on its own terms, but therefore must specify  $c$  as an output variable. To conveniently bypass this effort, we briefly introduce two additional operations that will help us.

**Output Hiding.** Both parallel composition and mode refinement result in an automaton that unites the output variables of their respective automata. Sometimes its good practice to *hide*

certain output variables, when automata are composed sequentially (i.e., an output is connected to an input; cf. [Definition 3.8](#)). Formally, we write  $\mathcal{A} \setminus \{x\}$  to express that output variable  $x$  is not exported to the outside world.

**Input Binding.** With input binding, we instantiate automata with a concrete value for one input variable of an automaton  $\mathcal{A}$ . Semantically, this is equivalent to a parallel composition with a *dummy automaton* that models the needed behavior. For example, imagine that  $\mathcal{A}$  has as input  $c$ , which we want to permanently bind to value 1. Then we can apply  $\mathcal{A}_{aux} \parallel \mathcal{A}$ , where  $\mathcal{A}_{aux} = \langle \{q\}, \{q\}, \emptyset, \{c\}, \emptyset, \epsilon, \text{Ctrl} \rangle$  with  $\text{Ctrl}(q) = \{c := 1\}$ . Formally, we write  $\mathcal{A}[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$  to denote that variables  $x_1, \dots, x_n$  are bound to values  $v_1, \dots, v_n$ .

Finally, we extend the grammar of  $\text{SL}_{\text{disc}}$  to enable mode refinement and parallel composition on the syntactic level. For that, we view refining automata as *submodes* that become active when the respective *supermode* is entered, and are deactivated when the respective supermode transits to a different mode.<sup>10</sup> We introduce the following new keyword for communicating with submodes.

**Definition 3.12: smode ( $\cdot$ )**

The grammar of  $\text{SL}_{\text{disc}}$  is extended by keyword **smode** ( $\cdot$ ), where  $S$  is a statement of the original syntax for  $\text{SL}_{\text{disc}}$  and  $S'$  is a statement of the new syntax:

$$S' ::= S \mid S ; \text{smode } \underbrace{M_1[\cdot] \setminus \{\cdot\} \parallel \dots \parallel M_n[\cdot] \setminus \{\cdot\}}_{\text{binding + hiding}}$$

$$S ::= \dots \text{ (as in Definition 3.2)}$$

Again,  $M_1, \dots, M_n \in M$  denote abstract placeholder symbols, where  $[\cdot] \setminus \{\cdot\}$  denotes the successive application of *input binding* and *output hiding*, and  $\parallel$  retains its meaning of parallelism. These symbols can be resolved with a given binding  $\xi_{\text{smode}} : M \rightarrow \mathcal{HMA}$ . To form the complete controller logic, submodes  $M_1, \dots, M_n$  need to be resolved in the following scheme, where  $q \in Q$  is the supermode in  $\mathcal{A}$  that applies the **smode** ( $\cdot$ ) command.

**Algorithm 3.1** (Scheme for Resolving **smode** ( $\mathcal{A}_1 \parallel \dots \parallel \mathcal{A}_n$ ) of mode  $q$ ).

1. Identify hybrid mode automata  $\mathcal{A}_1, \dots, \mathcal{A}_n$  with  $(M_i, \mathcal{A}_i) \in \xi_{\text{smode}}$ . If  $\xi_{\text{smode}}$  is undefined for  $M_i$  with  $i = 1 \dots n$ , **smode** cannot be resolved.
2. Reduce all  $\mathcal{A}_i[\cdot] \setminus \{\cdot\}$  to  $\mathcal{A}'_i$  by performing input binding and output hiding.
3.  $\mathcal{A}'_1, \dots, \mathcal{A}'_n$  must be pairwise composable, which can be checked by applying [Definition 3.7](#). If not, **smode** cannot be resolved.
4. Build a new hybrid mode automaton by pairwise parallel composition:

$$\mathcal{A}_{\text{sub}} = (\dots (\mathcal{A}'_1 \parallel \mathcal{A}'_2) \parallel \mathcal{A}'_3) \parallel \dots \parallel \mathcal{A}'_n).$$

<sup>10</sup>This view for transition systems is most prevalent for statecharts and hierarchical state machines [[Harel 1987](#); [Harel et al. 1998](#)], which ideally even enjoy the *Liskov substitution principle* [[Liskov et al. 1986](#)] for states (i.e., informally, substates do not violate the purpose of the superstate).

5. Refine supermode  $q$  by  $\mathcal{A}_{sub}$ :

$$\mathcal{A}_{res} = q \blacktriangleright \mathcal{A}_{sub}.$$

□

If the final mode refinement is admissible (which, again, can be easily checked; see [Definition 3.11](#)), then  $\mathcal{A}_{res}$  is a hybrid mode automaton that extends mode  $q$  with the combined behaviors of  $\mathcal{A}_1, \dots, \mathcal{A}_n$ . One peculiarity we ignored is that  $\mathcal{A}_1, \dots, \mathcal{A}_n$  can themselves define submodes with `smode` ( $\cdot$ ). There exist two solutions. First, before entering step 3, these submodes can themselves be recursively resolved, such that the resulting  $\mathcal{A}'_i$  is free of modes that use `smode` ( $\cdot$ ). Second, at step 3, submodes of  $\mathcal{A}'_1, \dots, \mathcal{A}'_n$  can be combined in the process by using parallel composition and the resolution scheme above can be applied continuously until no more submodes remain. Both solutions are semantically equivalent.

### 3.3. Syntax and Semantics of Skills and Skill Graphs

In the previous section, we introduced language SL and hybrid mode automata as a means to describe the behavior of hybrid skills. In this section, we finally focus on the formalization of skills and skill graphs, where hybrid mode automata and assume-guarantee interfaces play a key role. The goal is to develop a framework that is *modular* and allows to maximize *reuse* of already implemented elements. In particular, hardware and software skills are only described by their interfaces. We proceed with the formalization of skills and skill graphs in [Section 3.3.1](#). Afterwards, we present a sound technique to *compose* skill graphs in [Section 3.3.2](#), which allows to develop skill graphs (i.e., their represented maneuvers) in isolation and subsequently to combine them automatically.

#### 3.3.1. Formalization

Now that we have all the ingredients together, we can finally give formal definitions for skills and skill graphs. For this, we first assume that there exists a universe of unique identifier symbols `Name`, by which we may distinguish different skills. Moreover, each skill is *typed* with a type of the domain

$$\text{Type} = \{\text{behavior, action, perception, planning, sensor, actuator}\}$$

Types correspond to a skill's responsibility, as described in [Section 3.1](#), but also regulate, which skills in a skill graph can build a relationship. We proceed with the formalization of skills and skill graphs separately.

#### Formalization of Skills

When focusing only on the design of the controller logic, skills can be described by only four ingredients, namely (1) a unique name, (2) a type, (3) assumptions and guarantees over formulas in first-order logic with real arithmetic, and (4) a behavior that, given the assumptions, should enforce the guarantees. Formally, we provide the following definition.

**Definition 3.13: Syntax of Skills**

A **skill** is a tuple  $\langle \text{name}, \text{type}, \mathcal{I}, \mathcal{B} \rangle$ , where

- $\text{name} \in \text{Name}$  is a unique identifier symbol,
- $\text{type} \in \text{Type}$  is an associated type,
- $\mathcal{I} = \langle V^{\text{in}}, V^{\text{out}}, \phi^A, \phi^G \rangle$  is an assume-guarantee interface,
- $\mathcal{B}$  is an associated behavior over variables  $V^{\text{in}} \cup V^{\text{out}}$ . For **hybrid skills** (i.e.,  $\text{type} \in \{\text{behavior}, \text{action}\}$ ),  $\mathcal{B}$  is represented by a hybrid mode automaton

$$\mathcal{A} = \langle Q, q^0, V^{\text{in}}, V^{\text{out}}, \text{Trans}, \text{Dyn}, \text{Ctrl} \rangle.$$

For all other skills,  $\mathcal{B}$  is undefined for now.

We denote the **universe** of all skills by  $\mathcal{S}$ .

As our goal is to verify maneuvers represented by skill graphs (i.e., the controller logic) as early as possible, we only provide a concrete behavior for hybrid skills, which will become subject to verification. The behavior of all other skills is only represented by their assume-guarantee interfaces.<sup>11</sup>

**Example 3.6.** In Listing 3.2, we show an excerpt of the textual description of action skill **On** from the thermostat example (see Section 3.1.2). This skill is an isolated description of the behavior of the thermostat, when it is turned on. As seen before, the discrete controller sets the heating power to a value greater than  $u$  and the single ODE models the expected increase in temperature. The hybrid mode automaton is described on the right side, where the discrete and dynamic control of each mode is described after using keyword **behavior** and a mode identifier. The textual description comprises the three elements controller, dynamics, and transitions.

<pre> 1 <b>skill</b> On : <b>Action</b> 2   <b>input variables</b>: 3     x, u, k : ℝ 4   <b>output variables</b>: 5     h : ℝ 6   <b>assumption</b>: 7     x ≤ u 8   <b>guarantee</b>: 9     x ≤ u ∧ <b>past</b>(x) ≤ x </pre>	<pre> <b>init</b> M<sub>0</sub> <b>behavior</b> M<sub>0</sub>:   <b>controller</b>:     <b>havoc</b> h; <b>assume</b> h &gt; u   <b>dynamics</b>:     {x' = K(h - x) &amp; x ≤ u}   <b>transitions</b>:     <b>to</b> M<sub>0</sub> <b>when</b> x ≤ u; </pre>
---	---

Listing 3.2: Excerpt of the textual representation of skill On.

Skill **On** as illustrated in Example 3.6 does not depend on any other skill and may therefore be analyzed in isolation. However, we also saw in the previous section that behavioral skills on a higher level serve as controllers for behavioral skills on a lower level. For instance, skill **Heater** depends on skill **On** (or at least a valid replacement). Although we provide means for describing the intended

<sup>11</sup>This simplification allows us to assume *idealized* sensor values, perception and planning results (all possibly with known error rates), which clearly must be implemented when deriving a concrete program. Nonetheless, it is surely possible to model hardware skills as hybrid automata, as demonstrated by Mitra [2021:Ch. 11] for a simple powertrain model.

behavior in an abstract fashion (e.g., Hoare triples), eventually such dependencies must be resolved with a concrete instantiation. Skill graphs formalize these dependencies by expressing which concrete skills should work together to represent the intended maneuver. Before we formalize skill graphs, however, we first explain the new role of Hoare triples and invocations in skills.

### The Role of Hoare Triples and Invocations in Skills

In [Section 3.2.1](#), we introduced Hoare triples and invocations as part of SL. However, both constructs originally target the complete scope of SL, whereas hybrid mode automata explicitly partition the behavior of modes into a *discrete* and *dynamic* part. For practical usage, this notion is refined to fit the needs of hybrid mode automata.

With Hoare triples, a developer has two options. First, we allow to explicitly abstract *complete* modes with a single Hoare triple. For this, we add keyword **abstract\_behavior** to signify this approach. A correct implementation is then a concrete mode itself. Second, Hoare triples can also be used at any position in the discrete controller part of a normal behavior. Consequently, this Hoare triple is only implementable by a concrete program of  $SL_{disc}$ . We do not offer an option to use Hoare triples in the dynamics part of a normal behavior.

Invocations can be used in either the discrete or dynamic part of a normal behavior. However, invoked *code fragments* must also be either a program of  $SL_{disc}$  or  $SL_{dyn}$ , respectively. We signify such fragments with prefixes *ctrl* and *dyn*. Furthermore, such fragments are shaped like procedures with arguments and – as reasonable in the context of deductive verification – can be equipped with a precondition (keyword **requires**) and postcondition **ensures**. We view these fragments as reusable code snippets, which also increases modularity and reuse. In [Listing 3.3](#), we exemplify both concepts by modeling skill **Heater** of the thermostat example.

---

<pre> 1  dyn ODE_off(K, x, l) 2    requires x ≥ l; 3    ensures x ≥ l ∧ <b>past</b>(x) ≥ x; 4    := {x' = -Kx &amp; x ≥ l} 5 6  skill Heater : Behavior 7    input variables: 8      x, u, k : ℝ 9    output variables: 10     h : ℝ 11   assumption: 12     l ≤ x ≤ u 13   guarantee: 14     l ≤ x ≤ u </pre>	<pre> init M<sub>On</sub> <b>abstract_behavior</b> M<sub>On</sub>:   ⟨x ≤ u⟩M<sub>On</sub>⟨x ≤ u ∧ h &gt; u ∧ <b>past</b>(x) ≤ x⟩[x, h] <b>transitions</b>:   to M<sub>Off</sub> when x ≥ u; <b>behavior</b> M<sub>Off</sub>:   <b>controller</b>:     <b>skip</b>;   <b>dynamics</b>:     <b>invoke</b> ODE_off(K, x, l);   <b>transitions</b>:     to M<sub>On</sub> when x ≤ l; </pre>
--	---

---

Listing 3.3: Initial version of skill Heater.

Mode  $M_{On}$  is declared as abstract and its behavior is described by a Hoare triple. Mode  $M_{Off}$  invokes code snippet  $ODE\_off(K, x)$ , which represents the corresponding ODE. Although mode  $M_{On}$  does not have a concrete implementation yet, the description of **Heater** is already enough to be verified.

## Formalization of Skill Graphs

We formalize skill graphs as directed acyclic graphs, where nodes correspond to skills and directed edges represent their dependencies. As hinted before, only skills with particular types can form valid parent-child relationships, which we illustrate in Figure 3.6 with a defined type hierarchy. The edges represent a *can-be-child-of* relationship. We omitted self-directing edges for readability, but they exist for any skill except hardware skills.

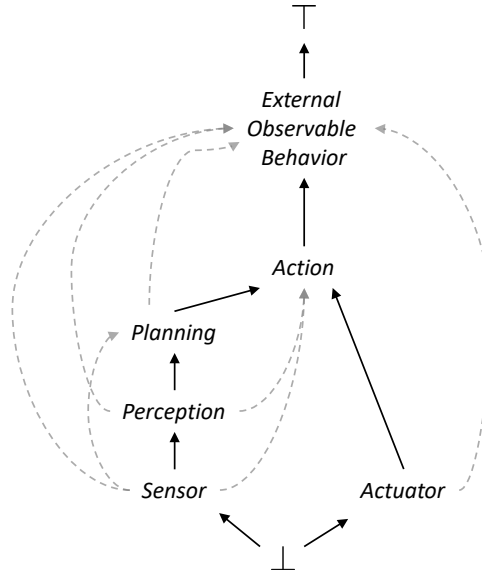


Figure 3.6.: Type hierarchy for skill graphs. Solid arrows illustrate the minimal hierarchical structure between skills, whereas grey arrows illustrate the transitive relation. Self-directed edges are omitted, but exist for all types except sensors and actuators.

Sensors can be child of any skill type except actuators. Conversely, actuators can only be child of hybrid skills, which is true for planning skills as well. Perception skills, however, can be child of planning skills. Furthermore, action skills can only be child of behavior skills. The bottom and top element informally represent that a well-formed skill graph needs to have sensors and actuators at the bottom and behavioral skills at the top. Next, we define the syntactic domain of skill graphs.

### Definition 3.14: Syntax of Skill Graphs

A **skill graph** is a triple  $\langle S, r, E \rangle$ , where

- $S \subseteq \mathcal{S}$  is a finite set of skills,
- $r \in S$  is the root skill,
- $E \subseteq S \times S$  is a set of directed edges between skills. An element  $(s, s') \in E$  consists of *predecessor* (or parent) skill  $s$  and a *successor* (or child) skill  $s'$ . If the context is clear, we use notation  $s \dashrightarrow s'$  to indicate that there exists an edge  $(s, s') \in E$ .

We denote the **universe** of all skill graphs by  $\mathcal{G}$ .



To make the modeling for developers more rigorous, we impose well-formedness constraints on skill graphs that prevent typical classes of ill-formed models. For instance, a skill graph always contains exactly one root skill  $r$  and the typing hierarchy as explained before must be respected. To do so, we need the *path* between two skills. Formally, a path of length  $l - 1$  is a (possibly empty) sequence of  $l - 1$  edges  $\pi_{s_1 \dashrightarrow s_l} = s_1 \dashrightarrow s_2, s_2 \dashrightarrow s_3, \dots, s_{l-1} \dashrightarrow s_l$ . A path between skills  $s, s'$  exists if  $\pi_{s \dashrightarrow s'}$  is non-empty (empty path denoted by  $\epsilon$ ), and does not exist otherwise.

#### Definition 3.15: Well-Formed Skill Graph

Let  $G = \langle S, r, E \rangle$  be a skill graph.  $G$  is well-formed if and only if:

- WF<sub>1</sub>: each skill  $s \in S \setminus \{r\}$  has at least one parent  $s_p \in S$  (i.e.,  $\exists s_p \in S : \pi_{s_p \dashrightarrow s}$  exists) and there always exists a path from  $r$  to  $s$  (i.e.,  $\forall s \in S \setminus \{r\} : \pi_{r \dashrightarrow s}$  exists),
- WF<sub>2</sub>: the root skill is of type `behavior` and each edge  $s \dashrightarrow s'$  respects the type hierarchy depicted in [Figure 3.6](#),
- WF<sub>3</sub>:  $G$  is acyclic (i.e., for  $s, s' \in S$ , if  $\pi_{s \dashrightarrow s'}$  exists, then  $\pi_{s' \dashrightarrow s}$  cannot exist),
- WF<sub>4</sub>: skills of type `sensor` have no input variables and skills of type `actuators` have no output variables,
- WF<sub>5</sub>: interfaces of sensor and software skills must be compatible (see [Definition 2.11](#)),
- WF<sub>6</sub>: for each variable  $v \in V_s^{\text{in}}$  of a non-actuator skill  $s$ , there must exist a path  $\pi_{s \dashrightarrow s'}$ , where  $v \in V_{s'}^{\text{out}}$  is defined. Conversely, for each variable  $v \in V_s^{\text{in}}$  of an actuator skill  $s$ , there must exist a path  $\pi_{s' \dashrightarrow s}$ , where  $v \in V_{s'}^{\text{out}}$  is defined,
- WF<sub>7</sub>: an output variable of a sensor or software skill is not allowed to be an output variable of a hybrid skill,
- WF<sub>8</sub>: for all  $s \in S$ , submodes must be resolvable (see [Algorithm 3.1](#)),
- WF<sub>9</sub>: for each skill  $s \in S$ ,  $\bigwedge_{s'} \phi_{s'}^G \rightarrow \phi_s^A$  must be satisfiable, where  $\phi_{s'}^G$  is the guarantee of any software skill  $s' \in S$  on an existing path  $\pi_{s \dashrightarrow s'}$ .

The first four well-formedness constraints are straightforward. With WF<sub>5</sub>, we require that we can see the collection of sensor and software skills as one composite with only defined output variables. That is, software skills must not define output variables if they were introduced before on skills on a path. WF<sub>6</sub> enforces that each input variable is connected to an output variable. WF<sub>7</sub> enforces that hybrid skills cannot redefine output variables. Intuitively, this means that the composite of software and sensor skills and the composite of hybrid skills are both compatible. Note, however, that these two composites have different meanings; each sensor and each software skill should be viewed as a single isolated process, whereas the collection of hybrid skills can be viewed as a single running process due to our notion of *mode refinement*. This is why multiple hybrid skills are allowed to define the same output variables, where compatibility is checked on the level of hybrid mode automata. WF<sub>8</sub> enforces us to include all skills that are necessary to resolve submode dependencies.



Finally,  $WF_9$  expresses that assumptions of *dependent* higher-level skills (i.e., software and hybrid skills) must be respected by guarantees of lower software skills.

### 3.3.2. Composition of Skill Graphs

From the perspective of software engineering practices, the advantage of skill graphs is their modularity, as they decompose the overall maneuver into smaller and easier manageable tasks. To take advantage of this, a core idea is to design multiple skill graphs in isolation and to subsequently compose them with little effort. The benefits are manifold. First, for concrete cyber-physical systems (e.g., specific robots or vehicles), lower level skills are often shared between maneuvers. For instance, a skill that manages longitudinal dynamics will be part of many different maneuvers. Composing skill graphs therefore increases reusability. Second, this supports the distribution of modeling tasks in multi-team software development. Third, when safety guarantees can be transferred from smaller skill graphs to a larger one, then identification and location of modeling mistakes is improved. Finally, this targets *scalability* of the verification strategy by minimizing costly re-verification.

Our composition technique is inspired by *superimposition* of graph structures, where two graphs are recursively merged together by *overlapping* their substructures. Starting from the root skill of one of the skill graphs, identical skills at the same level can then be merged together to form a new skill graph. However, this process forces us to think about a common root skill. Typically, skill graphs do not share a common root skill, as they represent dissimilar behaviors. From the perspective of a developer, there are two obvious reasons for composing skill graphs, which we refer to as *parallel composition* and *planned composition*:

**Parallel Composition.** The maneuvers represented by the two skill graphs can run in parallel, which should be exhibited by a third skill graph.

**Planned Composition.** The maneuvers represented by the two skill graphs should not run in parallel, but should be controlled with a top-level controller to form a new maneuver.

An example for the first reason for road vehicles would be keeping a distance to a leading vehicle while also keeping the car inside the lane. An example for the second reason would be the *explore world* maneuver, where turning is only performed after the robot comes to a halt. The first class of composition can be accomplished automatically, whereas the second class requires additional manual effort by developing a new behavior. For a more comprehensive discussion, we introduce so-called *pure controllers*. Pure controllers are typically the method of choice for behavioral skills and are located at higher levels in the skill graph hierarchy. Instead of writing to output variables themselves, they merely define operating modes, evaluate the incoming data, and allow to transit to a different mode. Discrete controller code and dynamics are handled by implemented submodes.

#### Definition 3.16: Abstract Mode and Pure Controller

Let  $q$  be any mode and  $\text{Ctrl}_q \in \text{SL}_{\text{disc}}$  its discrete program.  $q$  is called **abstract** if its discrete controller is of the form  $\text{Ctrl}_q \doteq \{\text{skip}; \text{smode} \dots\}$ . A **pure controller** is a hybrid mode automaton with no dynamic systems (i.e.,  $\text{Dyn} = \emptyset$ ), where each mode is abstract. We call a pure controller with only one mode a **combinatorial controller**.

**Example 3.7.** Consider the explore world maneuver illustrated in Figure 3.3. The top-level behavior **Explore World** is modeled as a pure controller with three modes: [Drive], [Brake], and [Turn]. The pure controller decides when to switch modes, but each mode delegates the actual controller output to the underlying skill **Control Dynamics**. The thermostat system does not have a pure controller. Nevertheless, skill **Heater** could outsource mode [Off] to a new action skill to then be considered a pure controller.

In the following, we define both types of composition in an algorithmic fashion. The *parallel composition* of skill graphs is essentially a simple synthesis problem, where we automatically synthesize a new root for the original two top-level behaviors. The *planned composition* uses an already defined behavior skill as a new root.

**Algorithm 3.2** (Parallel Composition of Skill Graphs). Let  $G_1 = \langle S_1, r_1, E_1 \rangle$  and  $G_2 = \langle S_2, r_2, E_2 \rangle$  be two well-formed skill graphs. Their parallel composition  $G = G_1 \parallel G_2$  is given by the following process if the behaviors of  $r_1$  and  $r_2$  are compatible.

1. Define a new skill  $r = \langle \text{name}, \text{behavior}, \mathcal{I}_{r_1} \parallel \mathcal{I}_{r_2}, \mathcal{A}_r \rangle$ , where name is a unique name and  $\mathcal{A}_r$  is a combinatorial controller with  $\text{Ctrl}_{q^0} \doteq \{\text{skip}; \text{smode } r_1 \parallel r_2\}$ .
2. Define a new set of skills  $S = S_1 \cup S_2 \cup \{r\}$ .
3. Define a new set of edges  $E$ . For every  $s, s' \in S \setminus \{r\}$ , there exists an edge  $(s, s') \in E$  if  $(s, s') \in S_1$  or  $(s, s') \in S_2$ . Additionally,  $(r, r_1), (r, r_2) \in E$ .
4.  $G = G_1 \parallel G_2 = \langle S, r, E \rangle$  is a new skill graph.

□

**Algorithm 3.3** (Planned Composition of Skill Graphs). Let  $G_1 = \langle S_1, r_1, E_1 \rangle$  and  $G_2 = \langle S_2, r_2, E_2 \rangle$  be two well-formed skill graphs. Their planned composition  $G = G_1 \bullet_r G_2$  for a manually developed hybrid skill  $r$  is given by the following process.

1. The behavior of  $r$  is a pure controller that refers to root skills  $r_1$  and  $r_2$  in its **smode**  $(\cdot)$ -declarations.
2. Define a new set of skills  $S = S_1 \cup S_2 \cup \{r\}$ .
3. Define a new set of edges  $E$ . For every  $s, s' \in S \setminus \{r\}$ , there exists an edge  $(s, s') \in E$  if  $(s, s') \in S_1$  or  $(s, s') \in S_2$ . Additionally,  $(r, r_1), (r, r_2) \in E$ .
4.  $G = G_1 \bullet_r G_2 = \langle S, r, E \rangle$  is a new skill graph.

□

For the parallel composition, we can see that assumptions and safety guarantees are transferred to the new top-level behavior by parallel composition of the interfaces. For the planned composition, the interface is manually written. In the next section, we explain how skill graphs are verified. We will see that validity of individual skill graphs transfers to their parallel composition.

### 3.4. Modular Verification of Skill Graphs

We have now established a formalization of skill graphs based on assume-guarantee interfaces and hybrid mode automata that enables us to finally reason about *correctness* of skill graphs and constructing associated *proofs*. In its simplest form, we may try to prove that the behavior of a skill in isolation indeed adheres to its specification (see [Definition 2.10](#)). This gives us the following task for each skill  $s \in \mathcal{S}$  of interest with its behavior  $\mathcal{B}_s$  and environment  $\mathcal{E}$ :

$$\phi_s^A \wedge \mathcal{E} \rightarrow [\mathcal{B}_s] \phi_s^G. \quad (3.1)$$

The environment is simply an invariant that expresses known facts about variables in  $\mathcal{V}^{\text{global}}$ . For instance,  $u > l$  is a known fact about constants  $u$  and  $l$  of the thermostat example and therefore  $\mathcal{E} \models u > l$ . If the behavior would constitute a simple hybrid program, [Equation 3.1](#) would represent a valid  $d\mathcal{L}$  formula, and proving it based on  $d\mathcal{L}$ 's calculus would suffice. For SL and hybrid mode automata, we first have to translate  $\mathcal{B}_s$  into a valid hybrid program, which is straightforward based on the fact that SL and hybrid programs are closely related.

Besides verifying that each skill is valid, validity of a skill graph additionally requires that all assumptions of each skill are met. For instance, if a sensor or software skill only operates within specified error margins, hybrid skills of a valid skill graph must be able to work within these error margins. Assuming smaller errors would potentially require to replace sensor or software skills.

In [Section 3.4.1](#), we present how complete skill graphs are transformed into hybrid programs and how the overall verification problem can be expressed as a formula in  $d\mathcal{L}$ . In [Section 3.4.2](#), we demonstrate how correctness of skill graphs can be transferred to their parallel composition.

#### 3.4.1. Transformation to Differential Dynamic Logic ( $d\mathcal{L}$ )

To verify correctness of a hybrid skill (cf. [Equation 3.1](#)) using the calculus of  $d\mathcal{L}$ , we first translate its hybrid mode automaton to a syntactically valid hybrid program. Following the semantics of a hybrid mode automaton, the idea of the translation is to split the behavior into two alternating parts. First, a transition is taken to proceed to the next mode.<sup>12</sup> This step always happens instantaneously without advancing time. Second, the mode is executed, where first the discrete controller is executed (again instantaneously) and subsequently the dynamic system runs and advances time. This also means that discrete controller and dynamic system must be translated from SL to the syntax of hybrid programs. In the following, we present the translation scheme in a top-level manner, where we first explain how the hybrid mode automaton is translated, and afterwards how discrete controller and dynamic system are translated.

In [Figure 3.7](#), we present the translation of a hybrid mode automaton to a hybrid program. To identify modes, we add a new variable  $m$  to the hybrid program that represents the *current* mode and assign each mode  $q$  a unique id denoted  $\text{id}(q) \in \mathbb{N}$ . For simplicity, we assume that  $\text{id}(q^0) = 0$ . As usual, the initial mode is  $q^0$ . Afterwards, the behavior alternates between taking a transition and executing the current mode. Both actions are resolved using nondeterministic choice together with evaluating the current mode  $m$ . This guarantees that in each run only the discrete

<sup>12</sup>Note that this can also mean that the initial mode is not the first mode that is executed. However, the first transition that is taken is always an outgoing transition of the initial mode.

$$\begin{aligned}
\text{trans}_{\mathcal{HMA}}(\mathcal{A}) &= \{m := \text{id}(q^0); (\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}})^*\} \\
\text{transitions}_{\mathcal{A}} &\equiv \bigcup_{(q,g,q') \in \text{Trans}} ?(m = \text{id}(q) \wedge g); m := \text{id}(q') \\
\text{modes}_{\mathcal{A}} &\equiv \bigcup_{q \in Q} ?m = \text{id}(q); \text{past}_q; \text{mode}_q \\
\text{past}_q &\equiv x_1^- := x_1, \dots, x_n^- := x_n \quad \text{for all free variables in } q \\
\text{mode}_q &\equiv \{\text{trans}_{\text{SL}}(\overline{\text{Ctrl}}(q)); \text{trans}_{\text{SL}}(\overline{\text{Dyn}}(q))\} \\
\overline{\text{Ctrl}}(q) &= \text{Ctrl}(q)[x_i \mapsto x_i^-]_{\text{rhs}} \quad \text{for all free variables in } q \\
\overline{\text{Dyn}}(q) &= \text{Dyn}(q)[x_i \mapsto x_i^-]_{\text{rhs}} \quad \text{for all free variables in } q
\end{aligned}$$

Figure 3.7.: Translation of hybrid mode automata to hybrid programs.

controller of the respective mode is executed. Following the semantics of hybrid mode automata (see [Definition 3.5](#)), free variables are replaced by their past version, which has only an effect if output variables are not only written, but also read.

Function  $\text{trans}_{\text{SL}}(\cdot)$  takes an SL program and translates it to a valid hybrid program. Most constructs we define in SL are straightforward to translate. Only the assert statement and the Hoare triple need more care. For the assert statement, we add a new unique variable called `assert`. In the beginning, we assume that `assert` will be equal to 0. If an assert statement cannot be satisfied in the current run, we add statement `assert:=1`. In the proving process, we still check if `assert=0` holds in the final state. If that is not the case, we will not be able to prove correctness of the behavior.<sup>13</sup> For Hoare triples, we follow the semantics of SL by first asserting the precondition, then applying nondeterministic choice to variables of the frame, and finally assuming the postcondition. In [Figure 3.8](#), we present the final translation of statements of SL to statements of hybrid programs.

In the following, we formalize the generation of verification conditions to verify hybrid skills in isolation and to assert the correctness of complete skill graphs.

## Verification Condition Generation

To verify a skill in isolation, we require that the contained hybrid mode automaton is free of submodes (see [Algorithm 3.1](#)). We may further assume that for resolving all submodes, we can refer to a public *repository* of skills, such that a skill is only verifiable if all submodes lead to existing skills and can indeed be resolved. Clearly, in the context of a skill graph, we have then to show that the exact same skills are also part of that skill graph. Then, for a hybrid skill  $s = \langle \text{name}, \tau, \mathcal{I}, \mathcal{A} \rangle$ , [Equation 3.1](#) is rewritten as

$$\phi_s^A \wedge \mathcal{E} \wedge \text{assert} = 0 \rightarrow [\text{trans}_{\mathcal{HMA}}(\mathcal{A})]\phi_s^G \wedge \text{assert} = 0, \quad (3.2)$$

<sup>13</sup>It would also be possible to directly extend the syntax of hybrid programs with a new `assert` keyword and to extend  $d\mathcal{L}$ 's calculus with a new rule  $[\text{assert } Q]P \leftrightarrow (Q \wedge P)$ , where  $Q$  is a logical formula and  $P$  a formula in  $d\mathcal{L}$ . However, it is more convenient (and trustworthy) to apply  $d\mathcal{L}$  as-is.

$$\begin{aligned}
\text{trans}_{\text{SL}}(S_1; S_1) &= \text{trans}_{\text{SL}}(S_1); \text{trans}_{\text{SL}}(S_1) \\
\text{trans}_{\text{SL}}(x := \theta) &= x := \theta \\
\text{trans}_{\text{SL}}(\text{havoc } x_1, \dots, x_n) &= x_1 := *; \dots; x_n := * \\
\text{trans}_{\text{SL}}(\text{assume } H) &= ?H \\
\text{trans}_{\text{SL}}(\text{skip } H) &= ?\text{true} \\
\text{trans}_{\text{SL}}(\text{if}(H)\{S_1\}\text{else}\{S_2\}) &= ?H; S_1 \cup ?\neg H; S_2 \\
\text{trans}_{\text{SL}}(\text{assert } H) &= \{?\neg H; \text{failure} \cup ?H\} \\
\text{trans}_{\text{SL}}(\text{invoke } M) &= \text{trans}_{\text{SL}}(\xi(M)) \\
\text{trans}_{\text{SL}}(x' = f(x) \ \& \ H) &= t := 0; \{x' = f(x), t' = 1 \ \& \ H \wedge t \leq \text{ep}\} \\
\text{trans}_{\text{SL}}(\langle P \rangle M \langle Q \rangle [x_1, \dots, x_n]) &= \{\text{trans}_{\text{SL}}(\text{assert } P); \\
&\quad \text{trans}_{\text{SL}}(\text{havoc } x_1, \dots, x_n); \\
&\quad \text{trans}_{\text{SL}}(\text{assume } H)\} \\
\text{failure} &\equiv \text{assert} := 1
\end{aligned}$$

Figure 3.8.: Translation of programs in SL to hybrid programs.

where variable `assert` is expected to remain unchanged (i.e., no assertions are violated). Clearly, verifying each skill in isolation following Equation 3.2 is not ideal, as higher level skills in a skill graph may refer to lower level skills via `smode`  $(\cdot)$ . Consequently, some automata are potentially resolved more than once and essentially re-verified, which introduces undesired redundancy. Only proving the top-level skill suffices to establish correctness of a maneuver, but also does not take advantage of the modular structure of skills. To make the leap towards more sophisticated modularity, we first derive a technical result with respect to compositionality, and afterwards discuss how we can improve the verification process.

#### Theorem 3.3: Proof Rule – Split Verification Condition

Let  $s = \langle \text{name}, \tau, \mathcal{I}, \mathcal{A} \rangle$  be a skill. Then Equation 3.2

$$\phi_s^A \wedge \mathcal{E} \wedge \text{assert} = 0 \rightarrow [\text{trans}_{\mathcal{HMA}}(\mathcal{A})]\phi_s^G \wedge \text{assert} = 0 \quad (3.3)$$

is valid if the following two formulas are also valid:

$$(\exists (q^0, g, q) \in \text{Trans} : \quad (3.4)$$

$$(\phi_s^A \wedge \mathcal{E} \models g) \wedge \phi_s^A \wedge \mathcal{E} \wedge \text{assert} = 0 \rightarrow [\text{past}_q; \text{mode}_q]\phi_s^G \wedge \text{assert} = 0)$$

and  $(\forall q \in Q :$

$$\phi_s^G \wedge \mathcal{E} \wedge \bigvee_{(q', g, q) \in \text{Trans}} g \wedge \text{assert} = 0 \rightarrow [\text{past}_q; \text{mode}_q]\phi_s^G \wedge \text{assert} = 0). \quad (3.5)$$

Intuitively, Equation 3.4 expresses the initial execution of the first mode after taking the first transition. Remember that a well-formed hybrid mode automaton must transition at each step, but the transition is unambiguous. Therefore,  $(q^0, g, q) \in \text{Trans}$  is guaranteed to exist and it is unique. Additionally, we can assume that  $\phi_s^A$  holds initially. Afterwards, safety guarantee  $\phi_s^G$  must hold indefinitely. Equation 3.5 evaluates the correctness of each mode in isolation. It can assume  $\phi_s^G$ , but not  $\phi_s^A$  anymore, as the initial condition might be violated by any previous executed mode. In the following, we proof this claim. Key steps are unrolling the loop presented in Equation 3.2 once and using  $\phi_s^G \wedge \mathcal{E} \wedge \text{assert} = 0$  as the loop invariant afterwards.

**Proof.** We prove Equation 3.3 based on  $d\mathcal{L}$  (cf. Section 2.2.2), where  $\Gamma \equiv \phi_s^A \wedge \mathcal{E} \wedge \text{assert} = 0$  and  $\Delta \equiv \phi_s^G \wedge \text{assert} = 0$ :

$$\begin{array}{c}
 \frac{(*)_2}{\Gamma \wedge m = \text{id}(q) \vdash [\text{past}_q; \text{mode}_q; (\dots)^*] \Delta} \text{ (eval)} \\
 \frac{\Gamma \wedge m = \text{id}(q) \vdash [\text{past}_q; \text{mode}_q; (\dots)^*] \Delta}{\Gamma \wedge m = \text{id}(q) \vdash [\bigcup? m = \text{id}(q'); \text{past}_{q'}; \text{mode}_{q'}; (\dots)^*] \Delta} (q = q') \\
 \frac{\Gamma \wedge m = \text{id}(q) \vdash [\text{modes}_{\mathcal{A}}; (\dots)^*] \Delta}{\Gamma \wedge m = 0 \vdash [\bigcup? (m = 0 \wedge g); m := \text{id}(q); \text{modes}_{\mathcal{A}}; (\dots)^*] \Delta} (\text{Unfold}) \\
 \frac{(*)_1 \text{ (eval)}}{\Gamma \vdash \Delta} \frac{\Gamma \wedge m = 0 \vdash [\bigcup? (m = 0 \wedge g); m := \text{id}(q); \text{modes}_{\mathcal{A}}; (\dots)^*] \Delta}{\Gamma \wedge m = 0 \vdash [\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}}; (\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}})^*] \Delta} (\exists(q^0, g, q)) \\
 \frac{\Gamma \wedge m = 0 \vdash [\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}}; (\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}})^*] \Delta}{\Gamma \vdash [m := 0; (\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}})^*] \Delta} (\text{iter. } [*]) \\
 \frac{\Gamma \vdash [m := 0; (\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}})^*] \Delta}{\Gamma \vdash [\text{trans}_{\mathcal{HMA}}(\mathcal{A})] \Delta} (\text{Unfold})
 \end{array}$$

$(*)_1$  represents the state where the loop is repeated zero times. We consider this state to be trivially true, as the safety guarantee must also hold initially for a maneuver. The evaluation of  $(*)_2$  corresponds to Equation 3.4. Afterwards, we proceed as follows, where  $J$  is a loop invariant.

$$\begin{array}{c}
 \frac{(*)_5}{J \wedge \bigvee_{(q', g, q) \in \text{Trans}} g \vdash [\text{past}_q; \text{mode}_q] J} \\
 \frac{(*)_3 \text{ (eval)}}{\Gamma \wedge m = \text{id}(q) \vdash J, \Delta} \quad \frac{(*)_4 \text{ (eval)}}{J \vdash \Delta} \quad \frac{\vdots}{J \vdash [\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}}] J} (\forall q \in Q) \\
 \frac{\Gamma \wedge m = \text{id}(q) \vdash J, \Delta \quad J \vdash \Delta \quad J \vdash [\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}}] J}{\Gamma \wedge m = \text{id}(q) \vdash [(\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}})^*] \Delta} (\text{Unfold}) \\
 \frac{\Gamma \wedge m = \text{id}(q) \vdash [(\text{transitions}_{\mathcal{A}}; \text{modes}_{\mathcal{A}})^*] \Delta}{(*)_2} (\text{loop})
 \end{array}$$

We choose  $J = \Delta \wedge \mathcal{E}$  for the loop invariant.<sup>14</sup> Then,  $(*)_3$  and  $(*)_4$  are valid, and  $(*)_5$  corresponds to Equation 3.5. For readability, we shortened the proof tree. For clarification, the term  $\bigvee_{(q', g, q) \in \text{Trans}} g$  stems from the fact that we consider each transition from a mode  $q'$  to  $q$  with guard  $g$ . For each transition, we get one, new proof obligation, where current mode  $m$  and guard  $g$  are part of the antecedent of that proof obligation. Consequently, we can unite all proof obligations with identical values for variable  $m$  using the inference rule for the logical or in the antecedent.  $\square$

The technical result of Theorem 3.3 enables us to think about how verification conditions can be simplified even further. For *abstract modes* with empty dynamical systems, one consideration is to not resolve submodes, but to abstract them with corresponding assume-guarantee

<sup>14</sup>In general, this choice for  $J$  may not always succeed, which is why the implication in Theorem 3.3 is only unidirectional.

contracts. In particular, submodes of a skill are associated with skills and therefore the respective interfaces are known. We have also defined composition for assume-guarantee interfaces (see [Definition 2.11](#)), which we can exploit. For instance, consider a mode  $q$  with controller  $\text{Ctrl}_q = \{\text{skip}; \text{smode } M_1 \parallel M_2\}$ , where  $M_1$  and  $M_2$  can be traced back to interfaces  $\mathcal{I}_1$  and  $\mathcal{I}_2$ . We may then reformulate the controller as follows:

$$\text{Ctrl}_q = \{\text{skip}; \text{assert } \phi_1^G \wedge \phi_2^G \rightarrow \phi_1^A \wedge \phi_2^A; \text{assume } \phi_1^G \wedge \phi_2^G\}.$$

This also means that verification of  $q$  after the first transition reduces to the following formula if  $q$  is a mode of skill  $s = \langle \text{name}, \tau, \mathcal{I}, \mathcal{A} \rangle$ :

$$\models (\phi_s^G \wedge \mathcal{E} \wedge \bigvee_{(q', g, q) \in \text{Trans}} g \rightarrow (\phi_1^G \wedge \phi_2^G \rightarrow \phi_1^A \wedge \phi_2^A)) \wedge (\phi_1^G \wedge \phi_2^G \rightarrow \phi_s^G).$$

Abstract modes without ODEs are therefore particularly interesting, as the verification conditions reduce to formulas of first-order logic for these modes. For pure controllers, we can generalize this idea with the following corollary.

#### Corollary 3.1: Proof Rule – Modular Verification of Pure Controllers

Let  $s = \langle \text{name}, \tau, \mathcal{I}, \mathcal{A} \rangle$  be a skill and  $\mathcal{A}$  a pure controller. Furthermore, let  $\Psi_q^A$  and  $\Psi_q^G$  denote the assumption and guarantee, respectively, of the assume-guarantee interface of the resolved submode for mode  $q$ . Then,

$$\phi_s^A \wedge \mathcal{E} \wedge \text{assert} = 0 \rightarrow [\text{trans}_{\mathcal{H}, \mathcal{M}, \mathcal{A}}(\mathcal{A})] \phi_s^G \wedge \text{assert} = 0$$

is valid if the following two formulas are also valid:

$$\exists (q^0, g, q) \in \text{Trans} : ((\phi_s^A \wedge \mathcal{E} \models g) \wedge \phi_s^A \wedge \mathcal{E} \rightarrow \Psi_q^A) \wedge (\Psi_q^G \rightarrow \phi_s^G) \quad (3.6)$$

$$\text{and } \forall q \in Q : (\phi_s^G \wedge \mathcal{E} \wedge \bigvee_{(q', g, q) \in \text{Trans}} g \rightarrow \Psi_q^A) \wedge (\Psi_q^G \rightarrow \phi_s^G). \quad (3.7)$$

**Proof.** As  $s$  is a pure controller, we know that  $\text{Ctrl}_{\mathcal{A}}(q) = \{\text{skip}; \text{assert } \Psi_q^A; \text{assume } \Psi_q^G\}$  and  $\text{Dyn}_{\mathcal{A}} = \emptyset$ . Based on [Theorem 3.3](#), we get:

- (1)  $(q^0, g, q) \in \text{Trans}_{\mathcal{A}} :$   
 $((\phi_s^A \wedge \mathcal{E} \models g) \wedge \phi_s^A \wedge \mathcal{E} \wedge \text{assert} = 0 \rightarrow [\text{past}_q; \text{mode}_q] \phi_s^G \wedge \text{assert} = 0)$
- (2)  $\forall q \in Q :$   
 $(\phi_s^G \wedge \mathcal{E} \wedge \bigvee_{(q', g, q) \in \text{Trans}} g \wedge \text{assert} = 0 \rightarrow [\text{past}_q; \text{mode}_q] \phi_s^G \wedge \text{assert} = 0).$

To simplify the proof, we observe that  $\text{past}_q = x^- := x; \dots$  and substituting each occurrence of a free variable  $x$  in  $\text{Ctrl}_{\mathcal{A}}(q)$  with their previous edition  $x^-$  cancel each other out, which is why we can ignore it in the following proof. We proceed by deduction, where we use  $\Gamma$  and  $\Delta$  as the respective placeholders for the precondition and postcondition:



$$\begin{array}{c}
(*)_1 \\
\frac{\Gamma \vdash \neg \Psi_q^A \rightarrow [\text{assert} := 1][? \Psi_q^G] \Delta}{\Gamma \vdash [\{? \neg \Psi_q^A\}[\text{assert} := 1][? \Psi_q^G] \Delta} ([?]) \\
\frac{\Gamma \vdash [\{? \neg \Psi_q^A\}[\text{assert} := 1][? \Psi_q^G] \Delta}{\Gamma \vdash [\{? \neg \Psi_q^A; \text{assert} := 1][? \Psi_q^G] \Delta} ([;]) \\
\frac{\Gamma \vdash [\{? \neg \Psi_q^A; \text{assert} := 1 \cup ? \Psi_q^A\}][? \Psi_q^G] \Delta}{\Gamma \vdash [\{? \neg \Psi_q^A; \text{assert} := 1 \cup ? \Psi_q^A\}; ? \Psi_q^G] \Delta} ([\cup]) \\
\frac{\Gamma \vdash [\{? \neg \Psi_q^A; \text{assert} := 1 \cup ? \Psi_q^A\}; ? \Psi_q^G] \Delta}{\Gamma \vdash [\text{trans}_{\text{SL}}(\text{skip}; \text{assert } \Psi_q^A; \text{assume } \Psi_q^G)] \Delta} (\text{trans}_{\text{SL}}(\cdot)) \\
\frac{\Gamma \vdash [\text{trans}_{\text{SL}}(\text{skip}; \text{assert } \Psi_q^A; \text{assume } \Psi_q^G)] \Delta}{\Gamma \vdash [\text{mode}_q] \Delta} (\text{Unfold})
\end{array}$$

Branch  $(*)_1$  affirms that if precondition  $\Gamma$  cannot imply  $\Psi_q^A$ , then a failure is the consequence. Hence,  $\Gamma \rightarrow \Psi_q^A$  must hold. Branch  $(*)_2$  affirms that if we can assume  $\Psi_q^A$ , then  $\Psi_q^G$  implies the postcondition. Hence,  $\Psi_q^G \rightarrow \Delta$  holds. Together, we get:

$$(\Gamma \rightarrow \Psi_q^A) \wedge (\Psi_q^G \rightarrow \Delta),$$

which, when applied to formulas (1) and (2) for each mode, results in [Equation 3.6](#) and [Equation 3.7](#).  $\square$

A special case that we need in the following is [Corollary 3.1](#) applied to skills with combinatorial controllers. In this case, only one mode exists and the guard of its self-transition reduces to true due to the correctness constraints of hybrid mode automata. The two formulas of [Corollary 3.1](#) for a skill  $s \in \mathcal{S}$  and mode  $q \in Q_{A_s}$  to prove then reduce to:

$$\begin{aligned}
& \models (\phi_s^A \wedge \mathcal{E} \rightarrow \Psi_q^A) \wedge (\Psi_q^G \rightarrow \phi_s^G) \\
\text{and} \quad & \models (\phi_s^G \wedge \mathcal{E} \rightarrow \Psi_q^A) \wedge (\Psi_q^G \rightarrow \phi_s^G).
\end{aligned}$$

### 3.4.2. Compositional Verification

In the previous subsection, we addressed the modular verification of skill graphs and what it means to verify hybrid skills in isolation to assert their validity. Validity of an entire skill graph follows naturally from the validity of each individual skill.

#### Definition 3.17: Valid Skill Graph

Let  $G = \langle S, r, E \rangle \in \mathcal{G}$  be a well-formed skill graph.  $G$  is **valid** if and only if for all hybrid skills  $s \in S$  (type  $\in \{\text{behavior}, \text{action}\}$ ), formula

$$\phi_s^A \wedge \mathcal{E} \wedge \text{assert} = 0 \rightarrow [\text{trans}_{\mathcal{HMA}}(\mathcal{A}_s)] \phi_s^G \wedge \text{assert} = 0$$

is valid. We write  $\models_G s$  to express that  $s$  is valid in  $G$  and we write  $\models G$  to express that the entire skill graph is valid.

Now we can formulate the main theorem of this chapter. Essentially, this theorem expresses that if two skill graphs are provably correct and both represented maneuvers can run in parallel with-



out interfering, then we can combine them to a third skill graph that is *automatically* provably correct. Even stronger, we can infer a new proof, which is simply the conjunction of all the individual results, namely the proofs of individual skills (i.e., their union), proof of compatibility between the roots hybrid mode automata, and [Corollary 3.1](#).

#### Theorem 3.4: Parallel Composition of Skill Graphs Retains Validity

Let  $G_1, G_2 \in \mathcal{G}$  be two valid skill graphs and let the hybrid mode automaton of their roots be compatible. Then, their parallel composition  $G_1 \parallel G_2$  yields also a valid skill graph.

**Proof.** Assume that  $G = G_1 \parallel G_2 = \langle S, r, E \rangle$  with  $r = \langle \text{name}, \text{behavior}, \mathcal{I}_{r_1} \parallel \mathcal{I}_{r_2}, \mathcal{A}_r \rangle$  being the new root skill.  $\mathcal{A}_r$  is a combinatorial controller with the single mode  $q$  and therefore by definition also a pure controller. Applying [Corollary 3.1](#) results in the following two formulas:

$$\begin{aligned} (1) \quad & \models (\phi_r^A \wedge \mathcal{E} \rightarrow \Psi_q^A) \wedge (\Psi_q^G \rightarrow \phi_r^G) \\ (2) \quad & \models (\phi_r^G \wedge \mathcal{E} \rightarrow \Psi_q^A) \wedge (\Psi_q^G \rightarrow \phi_r^G). \end{aligned}$$

Skill  $r$  is constructed in a way, such that  $\phi_r^A \equiv (\phi_{r_1}^G \wedge \phi_{r_2}^G \rightarrow \phi_{r_1}^A \wedge \phi_{r_2}^A)$  and  $\phi_r^G \equiv \phi_{r_1}^A \wedge \phi_{r_2}^A$ . However, the submode of  $\mathcal{A}_r$ 's single mode is also the parallel composition of  $r_1$  and  $r_2$ . Consequently,  $\Psi_q^A \equiv (\phi_{r_1}^G \wedge \phi_{r_2}^G \rightarrow \phi_{r_1}^A \wedge \phi_{r_2}^A)$  and  $\Psi_r^G \equiv \phi_{r_1}^A \wedge \phi_{r_2}^A$ . It follows that  $\phi_r^A \equiv \Psi_q^A$  and  $\phi_r^G \equiv \Psi_q^G$ . This leads to:

$$\begin{aligned} (1) \quad & \models (\phi_r^A \wedge \mathcal{E} \rightarrow \phi_r^A) \wedge (\phi_r^G \rightarrow \phi_r^G) \equiv \text{true} \\ (2) \quad & \models (\phi_r^G \wedge \mathcal{E} \rightarrow \phi_r^A) \wedge (\phi_r^G \rightarrow \phi_r^G) \equiv \text{true}. \end{aligned}$$

Condition (1) is easy to see. The left conjunct of condition (2) reduces to  $(\phi_{r_1}^A \wedge \phi_{r_2}^A) \rightarrow (\phi_{r_1}^G \wedge \phi_{r_2}^G \rightarrow \phi_{r_1}^A \wedge \phi_{r_2}^A)$ , which is a valid formula.  $\square$

## 3.5. Case Study: Vehicle Follow Mode

In this section, we describe our tool support and demonstrate our concept on a case study of a *vehicle follow mode* (i.e., an advanced adaptive cruise control), where the goal of the host vehicle is to safely follow another vehicle with a specified distance and activated lane keeping assistance. In particular, we give an overview of our tool SKEDITOR in [Section 3.5.1](#). We explain how we model and verify the vehicle follow mode and present results in [Section 3.5.2](#). Finally, in [Section 3.6](#), we discuss threats to the validity of our approach.

### 3.5.1. Open-Source Tool Support

We implemented the modeling, analysis, and verification of skill graphs in a tool suite with the name SKEDITOR. The implementation is written in JAVA and based on Graphiti<sup>15</sup>, which is an Eclipse-based graphics framework for creating graphical domain-specific languages. This way, we are able

<sup>15</sup><https://www.eclipse.org/graphiti/>

to embed our framework into the Eclipse IDE and make use of all its features. The language for implementing skills is written in Xtext<sup>16</sup>, which is a framework for creating textual domain-specific languages. The initial version of this tool support was developed in the course of a bachelor's thesis [Belli 2018]. In Figure 3.9, we illustrate the key aspects of SKEDITOR that directly reflect the theoretical concepts we developed throughout this chapter, namely (1) realizing skills with hybrid mode automata and modeling skill graphs, (2) translating skill graphs to proof obligations in  $d\mathcal{L}$ , and (3) asserting the correctness of these proof obligations with KEYMAERA X.

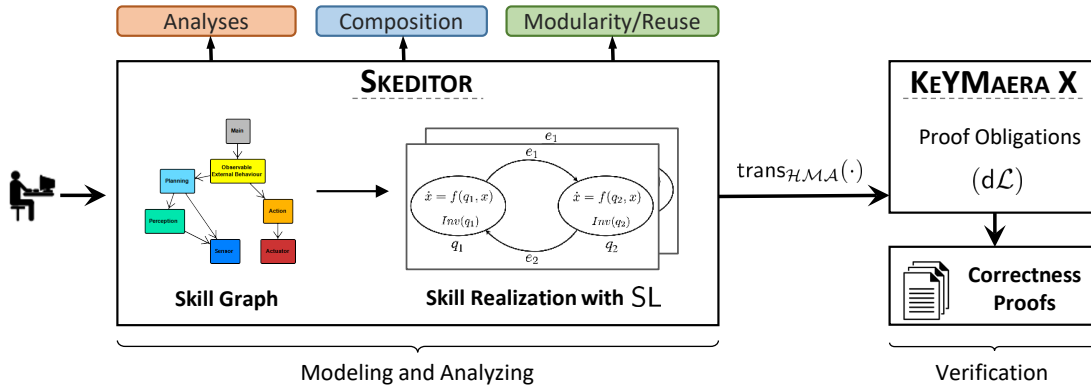


Figure 3.9.: Schematic overview of SKEDITOR.

Typical for Eclipse, users of SKEDITOR can create specific projects to model their skill graphs, which is done in a dedicated `*.sked` file. Skills and partial skill graphs can also be modeled in isolation to establish a *repository* of reusable skills and skill graphs, which can be imported to new skill-graph models via a dialog. Besides additional UI components, such as wizards for the composition mechanisms (see Section 3.3.2), several analysis components ensure that the well-formedness constraints of skill graphs and hybrid mode automata are satisfied. For instance, validity and compatibility of all interfaces in a skill graph are checked using the Z3 SMT solver [De Moura et al. 2008].

For realizing skills, the grammar of hybrid mode automata and SL is exactly implemented as described in Section 3.2 and Section 3.3. *Submodes* that are referred to by applying `smode` are resolved by using a skill's name. Resolution is successful if the respective skill is either found in the corresponding skill graph or available as an isolated module in the repository. SKEDITOR ensures that all names of skills in a project are unique. Importantly, SKEDITOR does not enforce all well-formedness constraints, as it provides the distinction between partial and complete skill graphs. For instance, if a submode cannot be resolved, a skill graph is labeled as *partial* (or incomplete), and can be completed in the future. The same holds true if input variables are not provided by lower-level skills.

Verification is accessible for partial and complete skill graphs. However, unresolved submodes prohibit a verification. To make the verification process more comfortable, KEYMAERA X is directly integrated in SKEDITOR and enables either the automatic or interactive verification of skill graphs. To verify a skill with KEYMAERA X, a user has the option to (1) either fully inline depended functionality to create the complete hybrid program (and verification problem in  $d\mathcal{L}$ ) of a maneuver or to (2) use abstraction mechanisms where possible, such as the composite interface for sub-

<sup>16</sup><https://www.eclipse.org/Xtext/>

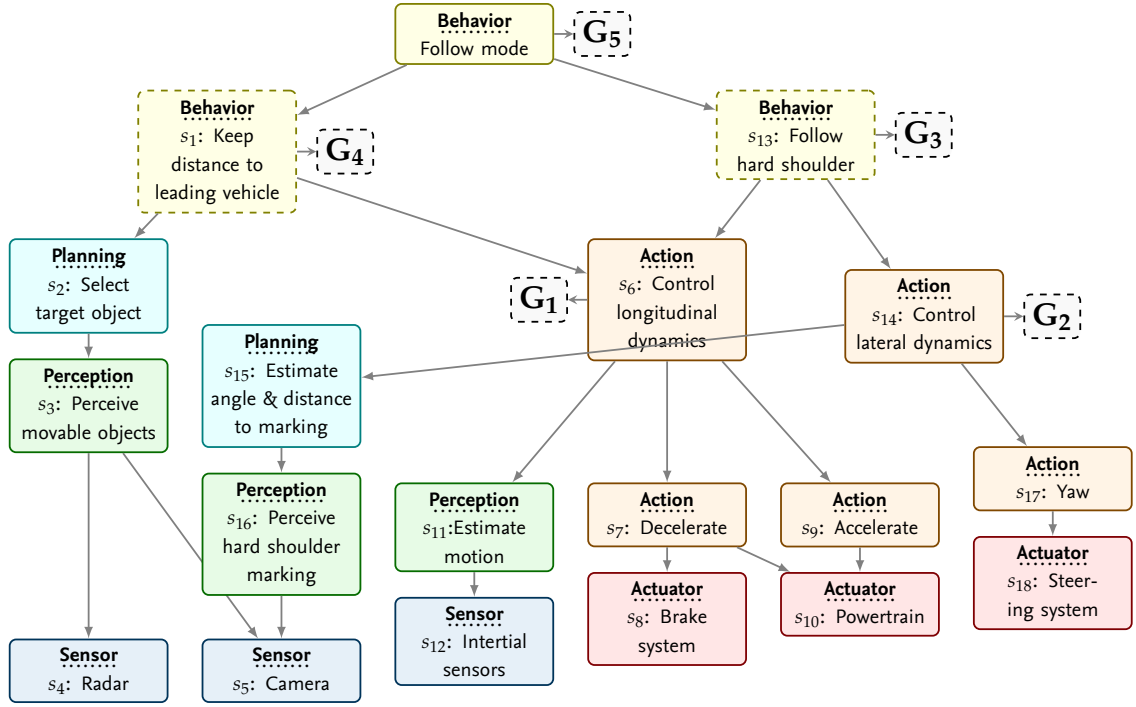


Figure 3.10.: Combined skill graph expressing the overall follow mode.

modes. If certain dependencies cannot be resolved, SKEDITOR automatically applies the abstraction mechanism even if inlining was applied. For instance, this is the case for Hoare triples, where only the translated abstraction is proven. A valid implementation for a Hoare triple must be provided manually in the current version of SKEDITOR.

### 3.5.2. Setup

We illustrate our skill-based modeling and verification approach with a *vehicle follow mode* performed on the hard shoulder that consists of two sub-behaviors. First, the host vehicle must keep a minimum distance to the lead vehicle to be able to perform an emergency brake without the possibility of a collision. Second, the host vehicle must not deviate too far from lane's center. This case study is adopted from Nolte et al. [2017], and was further extended in previous work [Knüppel et al. 2020a]. Essentially, the overall behavior combines an adaptive cruise control with lane keeping assistance for an unmanned vehicle operating on the hard shoulder. In Figure 3.10, we illustrate the complete corresponding skill graph.

The overall skill graph can be decomposed into five smaller skill graphs. In the following, we characterize these smaller skill graphs by their root skills.

**G<sub>1</sub> (Control longitudinal dynamics).** This substructure represents a typical cruise control with maximum velocity  $v_{max}$ . The skill itself controls the acceleration and braking force. Over a control variable  $c$ , it is possible to access the brakes and to decelerate. Although this is an action skill and the substructure represents a *partial* skill graph, a combinatorial controller as a new root skill would suffice to create a well-formed skill graph.

Skill	Type	Safety Requirement
<b>Follow hard shoulder</b>	Behavior	Combines safety guarantees of <b>Control lateral dynamics</b> and <b>Control longitudinal dynamics</b> .
<b>Control lateral dynamics</b>	Action	Lateral controller must guarantee overshoot of less than 25 cm and vehicle deviates from the center of the lane by at most half the lane width.
<b>Control longitudinal dynamics</b>	Action	Vehicle speed must not exceed $2.7 \text{ m s}^{-1}$
<b>Keep distance to leading vehicle</b>	Behavior	Vehicle must keep a minimum distance of 10 m to leading vehicle
<b>Follow mode</b>	Behavior	Combines safety guarantees of <b>Keep distance to leading vehicle</b> and <b>Follow hard shoulder</b> .

Table 3.2.: Excerpt of safety requirements for five hybrid skills of the vehicle follow mode.

**G<sub>2</sub> (Control lateral dynamics).** This substructure represents a lane keeping assist to keep the car inside the current lane. For this, it controls the yaw rate of the steering system. Analogous to **Control longitudinal dynamics**, this substructure represents a partial skill graph.

**G<sub>3</sub> (Follow hard shoulder).** This skill is a combinatorial controller and represents the parallel composition of the longitudinal and lateral dynamics. That is, the hybrid mode automaton of **Follow hard shoulder** consists of one abstract mode, which references both skills **Control longitudinal dynamics** and **Control lateral dynamics**. Informally, **Follow hard shoulder** controls the current velocity while simultaneously keeping the vehicle inside the lane.

**G<sub>4</sub> (Keep distance to leading vehicle).** This skill adds a distance measurement to the cruise control. That is, if the distance to the leading vehicle is too close, **Keep distance to leading vehicle** switches from cruise control to a mode that regulates the distance by decelerating the host vehicle.

**G<sub>5</sub> (Follow mode).** The overall behavior is a planned composition of the two sub-behaviors **Keep distance to leading vehicle**, which is active if the distance to the leading vehicle is getting too close and, **Follow hard shoulder**, which is active otherwise.

The informal safety guarantees for the skill graph of our case study are also adopted from Nolte et al. [2017]. We present an excerpt of informal safety guarantees for the five skills mentioned above in Table 3.2. Typically, safety requirements are given informally and must be translated to their formal counterpart. In particular, behavioral skills **Follow hard shoulder** and **Follow mode** are modeled as pure controllers and were developed with the already existing skill graphs in mind. Therefore, they combine the safety guarantees of their referenced skill graphs.

This incremental development of more complex maneuvers further emphasizes the high reuse potential of our framework that we developed throughout this chapter. For this case study, we are particularly interested in how much *verification effort* we are able to save during the development of all five skill graphs  $G_1, \dots, G_5$ . In particular, we concentrate on the following research question:

**RQ:** *To what extent can our proposed skill-based approach including its emphasis on compositional verification reduce verification effort?*

Skill Graph	Obligation	Proof Statistics		
		Step	Autom.	Time [s]
$G_1$	Monolithic	3,769	✓	22.1
$G_2$	Monolithic	7,223	✓	68.2
$G_3 = G_1 \parallel G_2$	Monolithic	16,924	(24)	312.2
	<b>Theorem 3.4</b>	-	✓	< 1.0
$G_4$	Monolithic	4,746	✓	29.3
	Submode abstraction	2,288	✓	15.2
$G_5 = G_3 \bullet_{FM} G_4$	Monolithic	22,018	(31)	485.9
	<b>Corollary 3.1</b>	-	✓	< 1.0

Table 3.3.: Comparison of the verification effort for skill-based compositional verification.

### 3.5.3. Results and Insights

In the following, we present results of our case study, where we compare verification effort of a monolithic modeling of all maneuvers with our approach that prioritizes abstraction and reuse. All measurements were conducted on an Intel i7-6600U CPU @ 2.60GHz with 12 GB RAM. For verification, we employed KEYMAERA X in version 4.7.3. Moreover, we used Z3 [De Moura et al. 2008] in version 4.6.0 as the internal solver for quantifier elimination, solving ODEs, and as regular SMT solver for checking validity of conditions. The *Follow Mode* case study is contained as example in our online repository of SKEDITOR.<sup>17</sup>

In Table 3.3, we summarize the verification results for the five skill graph introduced in the previous section. The *obligation* describes whether the respective skill graph was verified as a whole (i.e., monolithic), or a different condition was checked that leads to an equivalent verification outcome. The proof statistics report on (1) the number of automated proof steps produced by KEYMAERA X, (2) whether the obligation was verified automatically or needed a number of user interactions, and (3) the execution time for the automatic proof search (i.e., excluding time for user interaction). **Theorem 3.4** and **Corollary 3.1** were checked independent of KEYMAERA X, which is why no proof steps are reported.

As illustrated, the monolithic models of  $G_1$ ,  $G_2$ , and  $G_4$  could be verified automatically, whereas  $G_3$  and  $G_5$  needed a small number of user interactions. However, the parallel composition of  $G_1$  and  $G_2$ , which results in  $G_3$ , was automatically established by **Theorem 3.4**. Furthermore, the planned composition of  $G_3$  and  $G_4$ , which results in  $G_5$ , was likewise automatically established by **Corollary 3.1**. The three skill graphs  $G_1$ ,  $G_2$ , and  $G_4$  also needed the least number of proof steps. Whereas the monolithic model of  $G_4$  needed 4,746 proof steps, submode abstraction reduced the number of proof steps by approximately half. Proving correctness of the monolithic models of  $G_3$  and  $G_5$  took the longest and needed user interaction. In contrast, both the parallel composition for  $G_3$  and the planned composition for  $G_5$  with the newly introduced root skill FM were the most efficient, as only satisfiability of the respective conditions needed to be checked.

<sup>17</sup><https://github.com/AlexanderKnueppel/Skeditor>

The research question can therefore be answered positively. With respect to KEYMAERA X-4.7.3, most time was spent during quantifier elimination, which decides truth of real-arithmetic formulas. Our case study therefore highlights that reusing skills or complete skill graphs can significantly reduce the time spent during the automatic verification. Even more, user interaction could be completely avoided in our case study to establish skill graphs  $G_3$  and  $G_5$ . However, it is important to not overclaim this insight, as necessary user interaction is directly related to the *user-modeled* skill graph and how precisely it is specified.

### 3.5.4. Threats to Validity

Results of our evaluation are confronted with some threats to validity that we discuss in the following.

**Internal Validity.** The reported proof steps and measured execution time may greatly depend on the employed version of KEYMAERA X and the underlying solver for real arithmetic. Reproduction may therefore lead to different results, as KEYMAERA X is actively developed. However, the goal of this case study is to show that complete re-verification can be avoided, and that verification of monolithic models is costly. Under ideal circumstances, verification of a parallel or planned composition reduces to checking satisfiability of first-order logical conditions. The follow-mode maneuver in our case study constitutes such a non-trivial instance. We are therefore confident that our compositional verification approach allows to reduce verification effort independent of the employed version of KEYMAERA X.

The skill-graph model could be modeled differently, which may lead to a different outcome. We acknowledge that the results depend on the skill graph's shape and the implementation of all skills involved. Again, we used the case study as a non-trivial example to demonstrate our proof-of-concept. Moreover, interactive verification as applied in this evaluation is still challenging and takes considerable effort. Consequently, the trade-off between complexity of a case study and the feasibility of verifying it must be considered.

**External Validity.** Our verification results may not generalize to other maneuvers or application domains, as we only evaluated a single case study. The goal of our evaluation was to emphasize that our theoretical considerations of this chapter translate to the verification of non-trivial models. Moreover, the conditions we use that enable the application of [Theorem 3.4](#) and [Corollary 3.1](#) are not tied to a specific maneuver or application domain. In particular, the automotive domain provides typically only a small number of actuators (e.g., powertrain for braking and accelerating, and steering system), which quickly leads to interference of different maneuvers and consequently limits parallel composition. We expect that other application domains, such as robotics or multi-copters, will benefit even further from our theoretical and practical considerations.

## 3.6. Discussion

Besides the demonstration of our concept on the *vehicle follow mode* controller, we discuss fundamental choices we made in the formalization and implementation of skill-based mod-



eling and verification in the following. Focal points are language design, verification strategy, verification properties, and limitations.

**Minimalist Language Approach.** Instead of relying on general-purpose languages, we decided to design our own domain-specific language for implementing skills, where the language itself is inspired by labeled transition systems and hybrid programs. The reason for this is twofold. First, our primary focus at design time is to verify that maneuvers respect specific safety guarantees. The programming concepts we combined in language SL are completely sufficient to *model* any kind of controller logic that we intend to verify. In particular, sensors, perception skills, and planning skills are only described by their specification. It can be argued that the underlying software of these functions is typically more difficult to implement (e.g., due to uncertainties, computer vision algorithms, or machine learning approaches), which we can ignore at this stage in the development process. Moreover, simpler languages (e.g., without objects and heaps) are also inherently easier to verify, which improves the verification process as a whole. Of course, before an actual deployment, such controllers are typically re-implemented in some practical language to also consider non-functional properties, such as performance.

Second, hybrid mode automata focus more on realistic behaviors compared to hybrid programs. One goal of hybrid programs is to be able to represent a large family of possible controllers (i.e., by applying nondeterminism). Eventually, however, a very specific and deterministic controller must be realized. Although this saves verification effort, as each realized controller part of that family will be correct with respect to the family's specification, implementation effort is still high to eliminate nondeterminism. Hybrid mode automata remove nondeterministic choice as a language construct, which prevents this otherwise upcoming implementation effort. In summary, our choice was driven by easy modeling, more applicable verification, and also by an easier realization of a model in a general-purpose language.

**Theorem Proving versus Reachability Analysis.** In the context of formal verification of hybrid systems, reachability analysis is the predominant verification technique performed by model checkers [Benvenuti et al. 2014; Frehse et al. 2011; Frehse et al. 2004]. Typically, model checkers compute either an over-approximation of the set of reachable states or apply other kinds of heavy abstractions to reduce the size of this set. Afterwards, it is checked whether any of these states violates given safety properties. As usual, model checking hybrid systems suffers from the state-space explosion problem, which limits scalability. In contrast, we rely on a deductive technique (i.e., theorem proving) to formally verify safety requirements of modeled maneuvers. Consequently, we avoid such costly techniques for over-approximation and are also able to verify maneuvers at design time. Furthermore, we complete successful verifications with a proof, which increases possibilities for improving re-verification based on proof reuse, compositionality, and modularity. One disadvantage of theorem proving is its tendency to perform inadequately in case of a mismatch between specification and implementation. In the worst case, identifying the reason why a particular verification problem is not provable is almost impossible. Another disadvantage of theorem proving is that complex verification problems may require user interaction, as opposed to model checking, which is completely automatic. Ideally, a development process integrates both approaches:

model checking in the beginning mostly for debugging purposes, and theorem proving for certification and improved re-verification at the end.

**Specification Formalism.** We use  $d\mathcal{L}$  for specifying safety properties of skills. The reason is that  $d\mathcal{L}$  combines hybrid programs with a logical proof calculus, which aligns with our goal of deductive verification and generating proofs. We limit ourselves to *safety invariants* only, which must hold at all times during operation. The reason is that we consider these properties the most crucial ones that must be guaranteed at all cost and cannot be ignored (e.g., collision avoidance). Nevertheless, it is possible to extend the specification language to cover temporal properties over the execution of modes, as we already encode the overall behavior as an automaton and defined the execution semantics as traces. For instance, we could try to verify that the *explore world* maneuver lets our robot rotate on the spot only *until* it is safe to drive forward. This way, we not only verify that maneuvers are safe, but we can also verify whether specific driving decisions are modeled as intended.

**Model versus Real World.** Naturally, most real-world problems concerned with cyber-physical systems are very complex. This constitutes an adversary not only for the verification procedure, which is inherently plagued with scalability problems, but also for any modeler and developer, who must design these systems. An important question in our case is: *what is the trade-off between (a) modeling provably-correct maneuvers as presented in this chapter and (b) maneuver execution in a real-world setting?* This question will be the thematic anchor of the next chapter, where we make the leap from verifying maneuvers at design time to validating (i.e., simulating) them at run-time. Nonetheless, we base our verification model on the logical foundation for cyber-physical systems emphasized by, which is profound and has been successfully applied to a number of case studies [Jeannin et al. 2017; Loos et al. 2011]. As typical for theorem proving and deductive verification, success often depends on the complexity and sophistication of the specification.

## 3.7. Related Work

In the following, we discuss the role of skill-based modeling in the context of software engineering practices and elaborate on differences of our approach to related work on hybrid systems verification.

### Roots of Skills and Skill Graphs

In 1970, Mesarovic et al. [2000] authored a textbook about hierarchical and multilevel systems in large-scale industrial automation. The key essence was a formal theory about the coordination process that rigorously describes how specific levels in the hierarchy of such systems may interact with each other. Based on the work of Mesarovic et al. [2000], first concepts of bringing this hierarchical modeling into the realm of automated driving together with decomposing hierarchical functionality into subsystems (later to be known as *skills*) were introduced by Maurer [2000]. Siedersberger [2004] and Pellkofer [2003] extended these concepts further by introducing *skill networks*, which can be seen as the immediate predecessors of skill graphs. Skill networks are also directed and acyclic graphs with skills as nodes and dependencies as edges, but only consist of three hierarchical levels:



*actions*, *behaviors*, and *actuators*. Identical to our approach, behaviors and actions only differ in their fuzzy assessment of abstraction (i.e., whether a specific behavior is the focus of a skill network). In contrast to Maurer [2000], their focus was also on *skill realization* and *skill activation*, which they managed to describe with a mix of state machines and UML sequence diagrams. We adopted and generalized these considerations mathematically and practically by proposing hybrid mode automata for describing the activation of modes and language SL for implementing skills.

A decade later, skill networks were further extended, but mostly used to describe a maneuver’s architecture on a conceptual level. Bergmiller [2015] enhanced skills with performance metrics to enable monitoring the *quality* of active skills during operation. A corresponding performance model allows to derive decisions in the context of automated driving based on a current performance evaluation. Importantly, Bergmiller [2015] focused on by-wire-vehicles and did not consider perception or planning as part of skill networks explicitly. Finally, Reschka et al. [2015] proposed to incorporate skill networks in the item definition of ISO 26262 [2011] as a description for functional modules of driving tasks and their dependencies. They refined skill networks to the notion of *skill graphs* as adopted in this thesis. Besides sensor and actuator skills (among other data sinks, such as displays), they also considered perception and planning skills as an additional software layer.

Since then, skill graphs played a key role in publicly-funded research projects<sup>18</sup> combining model-based software engineering practices and automated driving [Nolte et al. 2017; Bagschik et al. 2018]. We contribute to this corpus of research by giving skill graphs a mathematical foundation that enables their analysis and verification at design time. Before, skill graphs were only used in a pen-and-paper fashion in the concept phase to describe and visualize functional dependencies. With SKEDITOR and our formal framework presented in this chapter, we bring skill graphs into the realm of real software engineering practices, where skill graphs can be modeled, implemented, verified, and even composed to create new maneuvers.

### Modeling and Verifying Hybrid Systems: State of Practice

Today, the most popular formal models to describe hybrid systems can be assigned to one of three archetypes, namely hybrid automata [Henzinger 2000; Alur et al. 1995; Alur et al. 1992], hybrid input/output automata [Lynch et al. 2003; Mitra 2021], and hybrid programs [Platzer 2008; Platzer 2010; Platzer 2012; Platzer 2018]. Related formalisms include hybrid dynamical systems [Goebel et al. 2012] or switched systems [Liberzon et al. 1999]. All models differ in their syntax and expressiveness. For instance, hybrid automata are easy to comprehend and put great emphasis on the dynamics of a system, but they are not compositional. That is, reasoning about smaller parts of a hybrid automaton is insufficient to reason about properties that correspond to the whole system. Therefore, Lynch et al. [2003] proposed hybrid input/output automata as a compositional evolution of hybrid automata. Hybrid programs constitute a simple and nondeterministic program notion for modeling cyber-physical systems that is also compositional. Besides the fact that hybrid programs can encode hybrid automata [Platzer 2008], they also allow to model *families* of cyber-physical systems due to their nondeterministic language constructs.

<sup>18</sup>One example of a publicly-funded research project is *Controlling Concurrent Change* (<http://ccc-project.org>), which focuses on model-based methods for integrating over-the-air updates at run-time. A second example is aFAS [Stolte et al. 2015], which aims at developing an unmanned and autonomously driving protective vehicle.

To implement skills, we adopt the notion of an automata-based approach together with hybrid programs as proposed by Platzer [2018], which has been applied successfully to numerous case studies including adaptive cruise control for road vehicles [Loos et al. 2011], airborne collision avoidance [Jeannin et al. 2017], and switched systems [Tan et al. 2021]. In contrast to pure hybrid automata, where each state represents a single assignment followed by evolution of the continuous system, our formalism allows to project larger programs onto states, similar to mode automata [Maraninchi et al. 1998; Maraninchi et al. 2003; Alur et al. 2001].

Another formalism specialized in modeling concurrent cyber-physical systems based on events is given by process calculi, such as hybrid communicating sequential processes [Chaochen et al. 1995; Liu et al. 2010] and HyPA [Cuijpers et al. 2005]. These formalisms are well studied and have been subject to theorem proving and model checking. In our approach, we did not focus on concurrency and communicating processes, but rather on single synchronous controllers.

An active area of research is *interoperability* between different modeling languages, formalisms, and tools. The tool HyST [Bak et al. 2015] aims at bridging the syntactical gap between a number of hybrid automaton modeling languages. Moreover, IPL [Ruchkin et al. 2018] is a domain-specific modeling language, which unites the verification of properties across different architectural views.

Verifying hybrid systems is mainly addressed by (1) model checking by reachability analysis, and (2) theorem proving. Reachability analysis aims at approximating the complete set of reachable states, which is compared to the set of desirable states of a given safety property. Although this technique focuses on full automation, it suffers from the state-space explosion problem. Well-known model checkers include SPACEx [Frehse et al. 2011], Flow\* [Chen et al. 2013], and C2E2 [Duggirala et al. 2015]. Most tools in this context focus on specific key aspects and differ in expressivity. Whereas SPACEx focuses on piecewise affine automata [Le Guernic 2009] (i.e., only linear dynamics), Flow\* and C2E2 both support verification of non-linear dynamics. In contrast to theorem proving, model checkers are less expressive and often highly specialized for a particular class of hybrid systems. Moreover, they incorporate complex data structures and algorithms, which also influences trust in the respective code base and the verification results.

As our focus is on deductive verification, we based our skill-based modeling approach on hybrid programs and  $d\mathcal{L}$ . Most proof assistants in the context of deductive verification, such as Coq [Coq Development Team. *The Coq proof assistant, 1989-2021*], AGDA [Agda Development Team. *The Agda wiki, 2007-2021*], LEAN [Moura et al. 2015], and ISABELLE/HOL [Nipkow et al. 2002], are based on a tactics language to carry out proofs. For hybrid programs, there exists Bellerophon [Fulton et al. 2017], which is integrated into KEYMAERA X and constitutes a library of powerful tactics for discharging obligations in  $d\mathcal{L}$ . Tailored to our context of skill-graph models, we formalized two proof rules (see Theorem 3.3 and Corollary 3.1) that enable us to reuse verification effort by simplifying the verification problem.

Kamburjan et al. [2022] presents a similar translation scheme to  $d\mathcal{L}$  to the one developed in this chapter for a hybrid extension of ABS called HABS. In contrast to skill-based modeling and verification, HABS is an *active objects programming language* with rich focus on concurrency and distributed computation. The goal is therefore to be less domain-specific, and to provide a more general approach for programming and verifying cyber-physical systems. The upside is that HABS can be used directly to develop all sorts of correct cyber-physical systems (including the interplay between independent components). That is, developers have the chance to implement the system resource-

efficiently while getting strong correctness guarantees, due to a focus on concurrency. The downside is that HABS requires much more expertise than the skill-based approach discussed in this chapter, and the integration into a software development process is not discussed. SKEDITOR provides a lower entry-barrier to the modeling of cyber-physical systems and focuses specifically on maneuvers. Due to its focus on model-based design, numerous specific analyses are available that would be difficult to apply in a too general setting. However, the downside is that the skill-based approach discussed in this chapter is less expressive than HABS (e.g., there exists no explicit focus on concurrency), which may hinder the optimal modeling of specific cyber-physical systems.

### 3.8. Chapter Summary

The goal of this chapter was to investigate how skill graphs can be modeled at design time, such that they become amenable for formal deductive verification. In particular, key questions were how do we specify safety properties of skills and how does a minimalistic programming language look like to proof safety compliance. For this, we specified skills with assume-guarantee interfaces and focused on a tight coupling with differential dynamic logic ( $d\mathcal{L}$ ) for deductive verification. We argue that our assume-guarantee approach enables modularity and reusability for skill implementations, and logical proofs provide strong guarantees, which maximize trust in the designed controller.

We proposed hybrid mode automata as an additional layer for modeling the behavior of skills based on the concept of operating modes. This is a natural follow-up to previous work on skills and skill graphs, where skills are conceptually realized by state machines. Nonetheless, hybrid mode automata constitute a much more precise mathematical notion that enables analyses and deductive verification of skill graphs at design time. In particular, we realize modes in a hybrid mode automaton with SL programs, which is a small extension of hybrid programs. Together with assume-guarantee interfaces, we formalized skills and skill graphs precisely, including important well-formedness constraints that can be statically checked by our tool support. Moreover, we presented two important compositions of skill graphs, namely parallel composition and planned composition. Both compositions are important in the context of software engineering practices, as they prioritize modular development and enable reuse. Alongside the theoretical ground work, we proved that compatible, valid hybrid mode automata can be composed and that validity transfers to the composite automatically. This result is invaluable to conquer scalability problems in formal verification.

We presented our tool support SKEDITOR for modeling and verifying skill graphs, which we used to evaluate our approach. SKEDITOR allows to translate skill graphs into hybrid programs with different degrees of abstraction (e.g., using contracts versus inlining) that can then be verified applying  $d\mathcal{L}$ 's deductive calculus. In a case study of a vehicle follow mode, we illustrated that both compositions are useful and that they effectively reduce proof effort. Most importantly, shared skills only needed to be verified once, which further emphasizes the modularity of our framework.

Finally, our considerations and case study illustrated that specifying and deductively verifying skill graphs is feasible at design time to rule out violating behaviors as early as possible. However, there still remains the question whether the specification correctly reflects the intention of the maneuver. An insufficient specification effectively renders the verification results useless. In the next chapter, we show how valid skill graphs can be refined to concrete implementations and how simulations can help to *validate* skill graphs.



## 4. Virtual Prototyping of Skill-Graph Maneuvers

*This chapter shares material with the ISoLA'20 paper “Scaling Correctness-by-Construction” [Knüppel et al. 2020b].*

In the previous chapter, we systematically discussed how to model, specify, and verify maneuvers of cyber-physical systems by combining skill graphs with differential dynamic logic [Platzer 2008]. As a result, a modeler is able to generate correctness proofs during the early design stage for the controller logic with KEYMAERA X [Fulton et al. 2015] to mathematically assert conformance of the respective maneuver and its specification. We have seen that our high-level abstraction of maneuvers combines comfortable modeling with scalable verification. The drawback of such high-level abstraction is, however, the presence of *abstraction gaps* when deriving concrete executables: (i) several abstraction mechanisms provided by our formal model, such as Hoare triples and non-determinism (see Section 3.2), must be resolved by end-users, (ii) hardware driver and software skills must also be implemented by end-users, which can lead to new bugs, (iii) correctness of the controller logic alone does not provide sufficient insights on the usefulness of the maneuver, and (iv) the modeled dynamics may not sufficiently reflect the real world. In particular, the last two gaps require us to develop concepts for *maneuver validation* at run-time, such as virtual simulation and monitoring.

In this chapter, we show how to derive implementations for virtual simulation from verified skill graphs, while maintaining a software engineering perspective that prioritizes *software correctness, reuse, and modularity*. We give a high-level overview of our verification and validation pipeline in Section 4.1. In particular, we introduce a component-based architectural framework called ARCHICORC [Knüppel et al. 2020b] as an intermediate layer between verified skill graphs and virtual simulation. ARCHICORC extends the principles of CORC [Runge et al. 2019a] (see Section 2.1.3) by structuring correct-by-construction programs into UML-style software components [UML 2 2017], which conceptualizes *correct-by-construction architectures*. All together, we provide means for formal specification and deductive verification from an early system’s model, over the architectural level to the source-code level. We identified this to be an underrepresented research area, which is worth to explore. In particular, for the first time, we integrate correctness-by-construction through ARCHICORC into an iterative development process. We present ARCHICORC’s component model in Section 4.2 and the corresponding tool chain in Section 4.3. In Section 4.4, we evaluate our verification and validation pipeline empirically. Finally, we discuss related work on the correctness-by-construction approach, component-based design with support for contracts, and end-to-end verification of cyber-physical systems in Section 4.5.

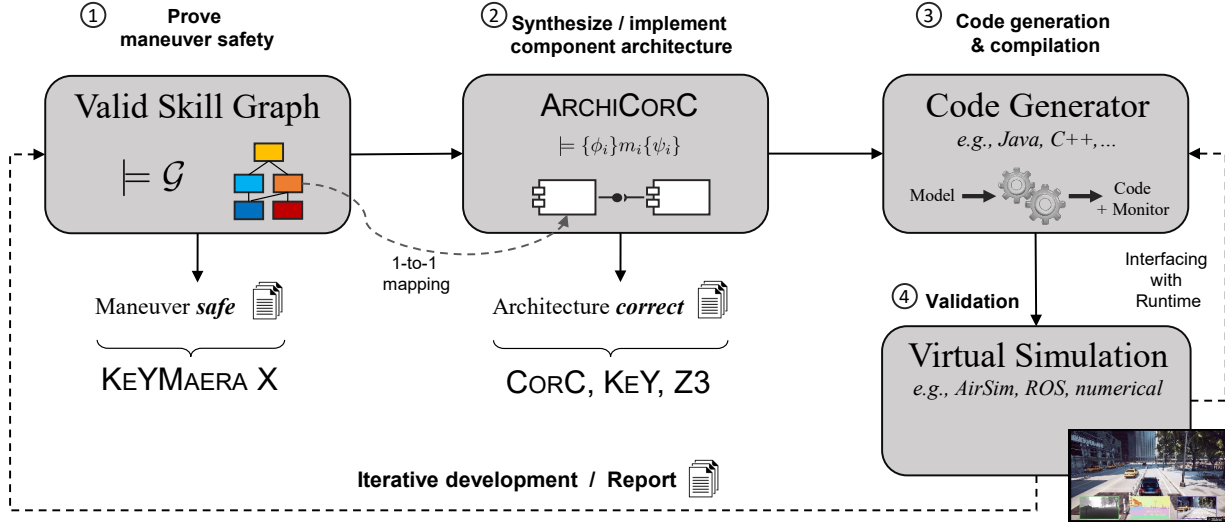


Figure 4.1.: Schematic overview of steps in the proposed verification and validation pipeline.

## 4.1. Overview of the Verification and Validation Pipeline

Before introducing the main concepts of ARCHICORC and how skill graphs can be translated for simulation, we give a high-level overview of the key ingredients of our verification and validation pipeline. We show a schematic overview in [Figure 4.1](#), which depicts an iterative development process consisting of four consecutive steps.

**Step 1: Proving Maneuver Safety.** The process begins with modeling and specifying the intended maneuver of a cyber-physical system as a skill graph (see [Chapter 3](#)). Then, a user translates the skill-graph model to differential dynamic logic ( $d\mathcal{L}$ ) and employs KEYMAERA X (either automatically or interactively) to generate correctness proofs. At this stage, we have asserted that the maneuver satisfies specific safety properties (e.g., collision freedom or maximum velocity) under a set of optimistic assumptions. In particular, software and hardware skills are only abstractly represented by their input variables, output variables, and specification. Eventually, these skills must be implemented and, to maintain safety, the correctness of their implementations with respect to safety properties must be assured. A different aspect is that the skill-graph models enjoy a high-level of abstraction, such as nondeterminism and Hoare triple specifications. All these aspects must be considered when deriving a concrete implementation in the next step.

**Step 2: Synthesizing and Implementing a Component Architecture.** The development process continues with a translation of the verified skill graph to a component-based architecture. We choose to focus on this intermediate layer, as component-based software engineering is usually a means to enable systematic reuse, rigorous abstraction from implementation details, and extensively focuses on decomposition [[Gössler et al. 2005](#)]. That is, building complex components from simpler ones is a key characteristic. In particular, we exploit ARCHICORC [[Knüppel et al. 2020b](#)], which builds on top of CORC [[Runge et al. 2019a](#)] (see [Section 2.1.3](#)) and enables a developer to establish a component-based *correct-by-construction architecture*. ARCHICORC components follow the UML notation [[UML 2](#)

2017] and define *provided* and *required* interfaces that comprise method signatures. In contrast to traditional component-based software engineering, method signatures  $m$  in ARCHICORC components are associated with a contract  $\{\phi\}m\{\psi\}$  and can therefore be either developed by CORC or become subject to post-hoc verification. To increase reuse, each software and hybrid skill in a skill graph is implemented by exactly one ARCHICORC component (i.e., with a one-to-one mapping) in a strictly defined manner. Hardware skills must be implemented by the respective simulation environment. As typical for software components, *composite* components allow component nesting, such that only the top-level component's specification is visible to the environment. For example, a planning skill for calculating the next waypoint on a trajectory may be mapped to a composite component providing methods for retrieving the calculated coordinates, and is eventually developed by composing numerous more components internally. Additionally, dependencies between skills in the original skill graph directly correspond to connectors between components, which can therefore be generated automatically. In summary, this process leads to a correct-by-construction architecture. The ARCHICORC component model is introduced in Section 4.2.

**Step 3: Code Generation and Compilation.** After software and hybrid skills are mapped to components and a component-based architecture is established based on the dependencies between skills, the architecture is translated to source code in a general-purpose programming language (e.g., JAVA or C++). Although the code generation itself is unverified per se, each part of this process is verifiable when following best practices. That is, behaviors of components ideally follow the correctness-by-construction approach and checking validity of connections between components (horizontally and vertically) reduces to satisfiability checks based on their interface specifications. In particular, behavior of software skills is typically developed manually starting with the top-level component's specification. For hybrid skills, KEYMAERA X provides means for generating C++ code of the controller [A. Müller et al. 2018b] when provided a  $d\mathcal{L}$  formula, which we use as inspiration for our tool chain. The generated controller code can be automatically translated to an underspecified CORC program (i.e., with missing intermediate specifications). With additional manual effort, this enables verification of the generated controller code.<sup>1</sup> In addition to the controller code itself, one particularity is the synthesizing of monitor code. Considering a  $d\mathcal{L}$  formula, the synthesized source code corresponds to the discrete part of the hybrid program, but ignores the precondition and safety guarantee. KEYMAERA X additionally integrates MODELPLEX [Mitsch et al. 2016], which synthesizes C++ code for monitoring violations of these conditions including the evolution constraint of the dynamical system. This way, assumptions and guarantees of the verified skill-graph model can automatically be transferred to the executed controller for manual or automatic inspection.

**Step 4: Validation.** The final step is to validate the derived executable on a set of scenarios. Similar to test cases in software testing, form and shape of scenarios depend on the analyzed functionality. For instance, the *explore world* maneuver (see Section 3.1.2) puts little restrictions on the actual scenario (i.e., the purpose is simply to guarantee collision freedom and otherwise drive freely), while

<sup>1</sup>Although the current version of CORC does not guide users in finding simple intermediate conditions automatically, there exist considerations to increase automation in the future. In particular, our generated controller code is free of loops, which should make the automatic identification of intermediate specifications simpler (e.g., using weakest precondition calculus [Dijkstra 1975]).



a lane keeping assistance for an automated vehicle should be simulated on actual roads with lane markings. For the visual simulation environments, we aim to integrate AirSim [Shah et al. 2018] by Microsoft, which is used for ground and air vehicles, and the robot operating system (ROS) together with gazebo, which is a mature framework and visualization environment for all kinds of robotic systems. As mentioned in step 2, the respective simulation environment must implement sensor and actuator skills to interface with the generated maneuver resulting from the previous step. Validation is typically performed manually, but supported by the automated monitoring of safety conditions. This allows to classify monitor violations as *passable* (i.e., violation does not have an impact on safety), *severe* (i.e., violation does have a high chance of impacting safety), or even *fatal* (e.g., vehicle collided with an object). Although there may still remain unsafe corner cases, validation is an important tool. For example, the derived controller is *untrusted*, as parts of the implementation are not verified during the first step. Additionally, it helps in identifying new assumptions that can be fed back into the development of the original skill graph.

As depicted in Figure 4.1, we exploit multiple provers and solvers to derive a correct executable from a skill graph: KEYMAERA X is used to verify controller correctness of the underlying hybrid system, CorC and the program verifier KEY [Ahrendt et al. 2016] are used to enable deductive verification on the derived implementation. Additionally, the SMT solver Z3 [De Moura et al. 2008] is employed to verify satisfiability of pure first-order logical conditions, such as contract compliance between connected components. All tools are mature and applied in a multitude of safety-related scenarios, which further increases trust in the obtained results of our verification and validation pipeline and the developed maneuvers.

## 4.2. The ARCHICORC Component Model

ARCHICORC [Knüppel et al. 2020b] is a tool that extends CorC with capabilities for UML-style component modeling. Although the idea is straightforward, such tool support increases the usability of CorC programs; it allows to systematically group correct-by-construction implementations into reusable components with defined interfaces to establish a *repository* of correct-by-construction libraries. In contrast to CorC, which focuses on independently developed algorithms without ownership and means for reuse, ARCHICORC aims at making the correct-by-construction approach available *at scale*. In particular, ARCHICORC consists of four main ingredients.

1. A **component and interface description language** to describe components, their required and provided interfaces (i.e., required and provided operations), and the connections between them. Operations can be annotated with specifications following the design-by-contract paradigm, where the concrete syntax is inspired by JML.
2. A **mapping** from operations of provided interfaces to either CorC programs or regular implementations.
3. A light-weight **formal reasoning framework** to check hierarchical and vertical compatibility of composed components, making the architecture correct-by-construction.



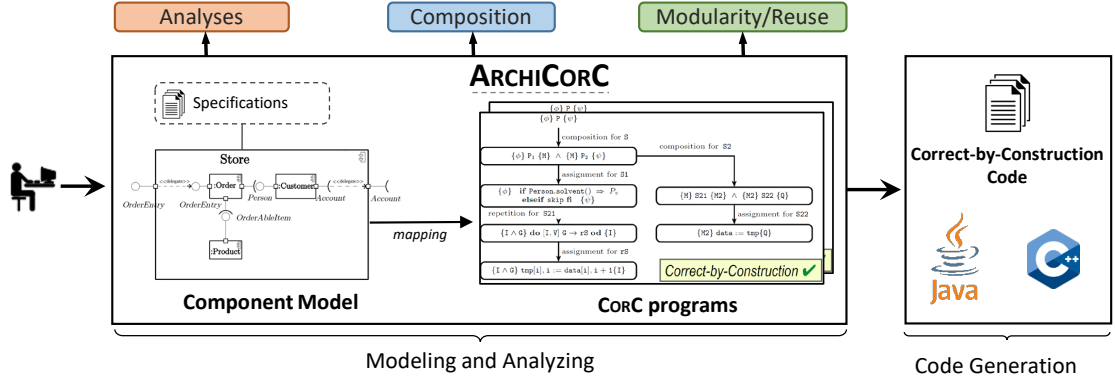


Figure 4.2.: Schematic workflow of ARCHICORC's development process.

4. A **code generator** that translates ARCHICORC components to (correct-by-construction) JAVA or C++ code.

In the following, we introduce the ARCHICORC development process and component model in more detail.

As typical for component-based modeling, *software components* combine a *behavioral model* (i.e., implementation details of the provided functionality) with an *interaction model* (i.e., how components communicate with each other) [Gössler et al. 2005; UML 2 2017]. In ARCHICORC, the behavior is represented by CORC programs, while interaction between components is based on classical *component interfaces*. Before formalizing parts of the ARCHICORC component model, we first sketch an overview of the ARCHICORC development process.

As illustrated in Figure 4.2, a user first creates a high-level structure of a component and associates required and provided component interfaces with it. Such interfaces comprise *method signatures* that are additionally annotated with a precondition and postcondition. As usual, components can be reused and hierarchically structured to form so-called *composite components* [UML 2 2017]. In a second step, method signatures of associated provided interfaces are mapped to *compatible* CORC programs. Informally, compatibility depends on two requirements: first, the method signature's specification of the CORC program must follow the *Liskov substitution principle* [Liskov et al. 1986] for preconditions and postconditions. That is, the precondition cannot be strengthened and the postcondition cannot be weakened in the CORC program with respect to the method signature. Second, method invocations in mapped CORC programs must be resolvable, which means that the corresponding method signature must be part of the component's *scope* (e.g., part of the required interface). Besides connecting a method signature to an already developed CORC program, a user can also try to derive a new correct-by-construction program starting from the method contract.

Valid compositions of components are guaranteed by formally reasoning about their contract compatibility. *Horizontal reasoning* checks whether connections between components are valid by proving that each component satisfies the assumptions of any component it is connected to. *Vertical reasoning* ensures that components can be substituted by other components satisfying again the Liskov substitution principle. In the final step, an ARCHICORC component is translated to correct-by-construction library code. In the following, we formalize ARCHICORC's component model and give examples with concrete syntax.

### 4.2.1. Interface Definition

Similar to UML-style component modeling, the interaction of ARCHICORC components is realized via *required* and *provided* interfaces that comprise method signatures. Additionally, each method signature  $m_i$  corresponds to a Hoare triple of the form  $\{\phi\}m_i\{\psi\}$  (i.e., is specified with precondition  $\phi$  and postcondition  $\psi$ ). We refer to such Hoare triples as *method contracts*. Identical to CORC, method contracts are expressed using a simplified version of the Java Modeling Language (JML) [Leavens et al. 2006].

#### Definition 4.1: Component Interface

A **component interface**  $I$  is a pair  $(M, F)$ :

- $M$  is a set of **method signatures**  $m_i \in M$  of the form  $\{\phi\}m_i(T\ a_1, \dots, T\ a_n) \rightarrow T\ r\{\psi\}$ , where  $T$  is a type universe (i.e., a placeholder for any particular type of interest), each method signature  $m_i$  maps a sequence of passed arguments  $a_1, \dots, a_n$  to a result value  $r$ , and  $\phi$  and  $\psi$  are the precondition and postcondition, respectively. We use shorthand  $\{\phi\}m_i\{\psi\}$  if the context is clear.
- $F$  is a set of typed variables called **fields**.

In contrast to the previous chapter, where variables and parameters were restricted to real values only, all arguments and the return value of a method signature, and fields may use any type permitted by an underlying type system. For the sake of presentation, however, we assume only primitive data types and standard typing rules (e.g., as presented by Pierce et al. [2002]). Explicit definitions of fields (i.e., variables) are permitted as a means for communication between methods. That is, fields are shared variables between method signatures. As will be formally defined later, components may implement specific interfaces (i.e., labeled as provided interfaces), while requiring functionality provided by other components (i.e., labeled as required interfaces). Considering required and provided interfaces of one component, fields of required interfaces are read-only by method implementations of provided interfaces (i.e., interpreted as getter methods), whereas fields of provided interfaces are both readable and writable by their corresponding method implementations.

**Example 4.1.** We revisit the thermostat example from the previous chapter (see Section 3.1.2), where two hybrid skills, namely **Heater** and **On**, are responsible for choosing a heating value  $h$  to keep temperature  $x$  between a lower limit  $l$  and an upper limit  $u$ . Furthermore, the behavior of skill **Heater** internally invokes the behavior of skill **On**. In the following, we exemplify a potential definition of a provided and required interface associated with the implementing component for skill **Heater** with the concrete syntax provided by ARCHICORC. We refer to the provided interface as `IHeaterP` and the required interface as `IHeaterR`. Then the concrete syntax of implemented interfaces according to Definition 4.1 may take the following form:

---

<pre> <b>archicorc_interface</b> IHeaterP {   <b>double</b> h;   //@ requires <math>\phi_{\text{Heater}}</math>;   //@ ensures <math>\psi_{\text{Heater}}</math>;   <b>void</b> ctrlStepHeater(<b>void</b>);   // ... } </pre>	<pre> <b>archicorc_interface</b> IHeaterR {   <b>double</b> h;   //@ requires <math>\phi_{\text{On}}</math>;   //@ ensures <math>\psi_{\text{On}}</math>;   <b>void</b> ctrlStepOn(<b>void</b>);   // ... } </pre>
--	--

---

The implementing component has the obligation to implement method `void ctrlStepHeater(void)`, but is allowed to invoke method `void ctrlStepOn(void)` implemented by another component. When invoked, both methods may update their respective variable `h`. For instance, executing `void ctrlStepOn(void)` may update `IHeaterR.h`, which can be read by `void ctrlStepHeater(void)` to update `IHeaterP.h`.

For the specification, it is tempting to propagate the precondition and postcondition of the corresponding skill to the component level. For instance, postcondition  $\phi_{\text{Heater}}$  could be defined as  $l \leq x \wedge x \leq u$  according to [Section 3.1.2](#). However, this is not a valid postcondition for `void ctrlStepHeater(void)`, as the change of `x` is a result of the underlying dynamic behavior and not a result of the method's implementation. Instead of specifying that the final implementation complies to the safety invariant, the goal is rather to specify that the implementation complies to the modeled controller that enforces the safety invariant. We postpone the derivation of component interface specifications for hybrid skills to [Section 4.3.1](#). For software skills, the precondition and postcondition can be propagated as-is.

In the following, we introduce two standard operations on component interfaces that increase flexibility of our formalization, namely *interface merging* and *method hiding*. In standard component-based modeling, each required interface  $I_{\text{req}}$  of a component must be connected to a provided interface of another component  $I_{\text{prov}}$ , such that all required method signatures can be resolved. In our formalism, multiple provided interfaces of different components may be merged together to match a required interface. Informally, component interfaces are *merge-compatible* if equally named methods are identical in signature and method contract, and equally named fields have the same type.

#### Definition 4.2: Merging of Component Interfaces

Let  $I = \{I_1, \dots, I_n\}$  with  $I_i = (M_i, F_i)$  be a finite set of merge-compatible component interfaces. Then, their **merging** is defined as

$$\text{merge}(I) = (\bigcup M_i, \bigcup F_i).$$

To be able to only provide a subset of methods of a component interface, we introduce an operator to hide methods and fields.

#### Definition 4.3: Hiding of Methods and Fields

Let  $I = (M, F)$  be a component interface,  $M' \subset M$  be a subset of method signatures of  $I$ , and  $F' \subset F$  be a subset of fields of  $I$ . **Hiding**  $M'$  and  $F'$  in  $I$  is defined as

$$\text{hide}(I, M', F') = (M \setminus M', F \setminus F').$$

### 4.2.2. Contract Refinement

We mentioned before that components may provide and require interfaces, and that method signatures of provided interfaces are eventually mapped to *method implementations*.<sup>2</sup> In particular, ARCHICORC is built on top of CORC, which is the primary program notation for such method implementations. Therefore, method signatures of our interface definitions (see Definition 4.1) and method contracts used as starting point in CORC programs (see Section 2.1.3) are based on the same formalism.<sup>3</sup> A mapping between both concepts is established again based on the Liskov principle and referred to as *contract refinement* [Hatchliff et al. 2012; Meyer 1988; Liskov et al. 1994]. Intuitively, given two method contracts  $c = \{\phi\}m\{\psi\}$  and  $c' = \{\phi'\}m'\{\psi'\}$ , method contract  $c'$  is a valid substitute for method contract  $c$  if  $c'$  *preserves* all behaviors specified by  $c$ .

Enabling contract refinement is desirable for at least two reasons. First, developers may link any CORC program to a provided method signature that constitutes a valid contract refinement. The alternative of only mapping identical method contracts can be seen as too restrictive from a software engineering perspective. Second, contract refinement can also be checked at the level of connections between provided and required interfaces. This way, specifications of mapped method contracts must not be identical, as long as the provided method contract preserves the behavior of the required method contract. Before giving a formal definition of contract refinement, we introduce the following predicate `equalSig` for two method signatures, where `type(·)` is a helper function returning the type of the passed variable name.

#### Definition 4.4: Matching Method Signatures

Let  $\{\phi\}m(T a_1, \dots, T a_k) \rightarrow T r\{\psi\}$  and  $\{\phi'\}m'(T b_1, \dots, T b_l) \rightarrow T r'\{\psi'\}$  be two method signatures. We say that the two method signatures  $m$  and  $m'$  **match** iff method name, number of arguments, argument types, and type of the return value are identical. We introduce predicate `equalSig` :  $M \times M \rightarrow \mathbb{B}$  stating whether two method signatures match as follows:

$$\text{equalSig}(m, m') \Leftrightarrow l = k \wedge m = m' \wedge \text{type}(a_i) = \text{type}(b_i) \wedge \text{type}(r) = \text{type}(r') \quad \forall i \in \{1, \dots, l\}.$$

Equality for method names (i.e.,  $m = m'$ ) means that both are lexicographically equal and equality for method arguments (i.e.,  $\text{type}(a) = \text{type}(b)$ ) means that  $a$  and  $b$  are equally typed. For sake of presentation, we do not apply the Liskov substitution principle on types, which would allow to define equality in a broader sense. We then define the refinement of method contracts as follows.

<sup>2</sup>We postpone the formal introduction of components and their interplay with component interfaces to the next subsection.

<sup>3</sup>Although the distinction is subtle, in contrast to method contracts, method signatures explicitly specify arguments and return types. In CORC's original work, there did not exist a concept of such signatures, as all variables were globally accessible. This changed with the introduction of the method call rule [Runge et al. 2019a; Bordis et al. 2020] (see Section 2.1.3). We will use method contract and method signature interchangeably.

**Definition 4.5: Method Contract Refinement**

Let  $c = \{\phi\}m(T a_1, \dots, T a_k) \rightarrow T r\{\psi\}$  and  $c' = \{\phi'\}m'(T b_1, \dots, T b_l) \rightarrow T r'\{\psi'\}$  be two method signatures. Method contract  $c'$  **refines**  $c$ , written  $c' \prec c$ , iff method signatures of  $m$  and  $m'$  match (i.e.,  $\text{equalSig}(m, m')$  is satisfied) and the following condition holds:

$$\models \phi \Rightarrow \phi'[b_i \setminus a_i] \wedge \psi'[\text{old}(b_i) \setminus \text{old}(a_i), r' \setminus r] \Rightarrow \psi.$$

As mentioned before, contract refinement also plays a role between required and provided interfaces. In particular, each method contract of a provided interface that is linked through a connector to a method contract of a required interface must satisfy **Definition 4.5** (i.e., be a valid contract refinement). With the following definition, we lift contract refinement to the level of interfaces.

**Definition 4.6: Component Interface Refinement**

Let  $I = (M, F)$  and  $I' = (M', F')$  be two component interfaces. Component interface  $I'$  **refines**  $I$ , written  $I' \prec I$ , iff

$$\forall m' \in M' : \exists m \in M, \text{ s.t. } \{\phi'\}m'\{\psi'\} \prec \{\phi\}m\{\psi\} \wedge F' \subseteq F.$$

**Example 4.2.** *Example 4.1* introduced interface `IHeaterR`. Consider the following interface `IONP` that refines `IHeaterR` iff  $\phi'_{\text{On}} \Rightarrow \phi_{\text{On}}$  and  $\psi_{\text{On}} \Rightarrow \psi'_{\text{On}}$  hold:

---

```

archicorc_interface IONP {
  double h;
  // ...possibly more fields

  //@ requires  $\phi'_{\text{On}}$ ;
  //@ ensures  $\psi'_{\text{On}}$ ;
  void ctrlStepOn(void);
  // ...possibly more method signatures
}

```

---

In component-based architectures, required interfaces of components are eventually connected to compatible provided interfaces of other components. In this context, component interface refinement is a convenient notion; it constitutes a compatibility check between connected, but non-identical provided and required interfaces. In particular, `IONP` may provide other methods and fields to the ones promised in its method signature, which increases reuse. We introduce components and connections between component interfaces in more detail in the next subsection.

### 4.2.3. Component Definition and Composition

A typical characteristic of component-based engineering is that components can be aggregated to build more complex components from simpler ones, eventually leading to a hierarchy of components. The lowest level of a component hierarchy consists of so-called *atomic* components that cannot be decomposed any further and are associated with actual computation. The other type of com-

ponent is represented by *composite* components with the sole purpose to hierarchically structure sub-components to reuse implemented functionality, but also to hide implementation details.

For describing computation, we denote by the abstract symbol  $\mathcal{P}$  the universe of possible *programs* (or implementations). Informally,  $\mathcal{P}$  refers to a sequence of statements in some programming model (e.g., a CORC program or a JAVA method). For the sake of presentation, we use the term *program* rather loosely if the context is clear. Furthermore, in alignment with design-by-contract, we say that a method signature  $\{\phi\}m\{\psi\}$  is *satisfied* by a program  $p \in \mathcal{P}$ , written  $p \models \{\phi\}m\{\psi\}$ , iff starting from a state where  $\phi$  holds, executing  $p$  results in a state that satisfies  $\psi$  and execution of  $p$  is guaranteed to terminate. We define an *atomic* component as follows.

#### Definition 4.7: Atomic Component

An **atomic component** is a triple  $(I_p, I_r, \text{Impl})$ , where

- $I_p$  is the (possibly empty) provided interface,
- $I_r$  is the (possibly empty) required interface,
- $\text{Impl} : M_p \rightarrow \mathcal{P}$  is a mapping that associates each method signature  $m \in M_p$  with a valid program in  $\mathcal{P} \cup \{\emptyset\}$ . An atomic component is **well-formed** iff  $\forall c = \{\phi\}m\{\psi\} \in M_p : \text{Impl}(m) \text{ exists } \wedge \text{Impl}(m) \models c$ .

We denote the **universe** of all atomic components by  $K_{\text{atom}}$ .

**Example 4.3.** We again consider the thermostat example to exemplify the concrete syntax provided by ARCHICORC for describing atomic components. Each definition of an atomic component starts with the keyword `atomic_component` and contains a list of provided and required interfaces. Additionally, provided interfaces can be mapped to CORC programs, which usually live in files with file ending `*.corc`.

---

```
atomic_component Heater {
  provide IHeaterP {
    ctrlStepHeater -> CorC/control_heater.corc
    // ...
  }
  require IHeaterR;
}
```

---

In this example, the CORC program `control_heater` provided by atomic component `Heater` has access to all method signatures defined in component interface `IHeaterR`. These methods have to be provided by another component implementing an interface that is compatible to `IHeaterR`.

In contrast to atomic components, composite components only indirectly establish a mapping from their provided interface to programs. Typical for UML-style components, they use *delegate connectors* that link external method signatures (i.e., required and provided interfaces) to particular interfaces of sub-components. Formally, we define such components as follows.

**Definition 4.8: Composite Component**

A **composite component** is a tuple  $(I_p, I_r, K_{sub}, Conn, Dele_p, Dele_r)$ , where

- $I_p$  is the (possibly empty) provided interface,
- $I_r$  is the (possibly empty) required interface,
- $K_{sub}$  is a finite set of *contained* sub-components (either atomic or composite),
- $Conn \subseteq \{k.I_p \mid k \in K_{sub}\} \times \{k.I_r \mid k \in K_{sub}\}$  is a set of connections between provided and required interfaces of sub-components in  $K_{sub}$ ,
- $Dele_p \subseteq \{k.I_p \mid k \in K_{sub}\}$  and  $Dele_r \subseteq \{k.I_r \mid k \in K_{sub}\}$  are sets of delegate connectors.

We denote the **universe** of all composite components by  $K_{comp}$ .

**Example 4.4.** Construction of composite components in ARCHICORC is similar to other UML-inspired component models [Haber 2016; Becker et al. 2009; Atkinson et al. 2008]. The thermostat example contains the two atomic components *Heater* and *On*, which may be composed to form a new composite component named *Thermostat*:

```
composite_component Thermostat {
  contains {
    atomic_component Heater, On;
  }
  provide IHeaterP;

  conn On.IOnP -> Heater.IHeaterR;
  dele Heater.IHeaterP -> this.IHeaterP;
}
```

As illustrated in the previous examples, component *Heater* provides interface *IHeaterP* and requires interface *IHeaterR*, while component *On* provides interface *IOnP*. The keyword **conn** is used to connect a provided interface with a required interface on the same level. In this example, interface *IOnP* is linked to interface *IHeaterR*. The keyword **dele** delegates a required or provided interface of a sub-component to the composite component's level. Here, interface *IHeaterP* is exposed to the environment.

In ARCHICORC, connections between components are always established as part of composite components. That is, even connected top-level components are contained explicitly in a specific root composite component. As the previous example illustrates, connectors link required interfaces of one component to provided interfaces of another one. Connections are valid if component interface refinement between them is satisfied (see Definition 4.5). Moreover, a *well-formed* composite component comprises only well-formed sub-components and must guarantee that internal connections and delegate connectors are all valid. We give the following formal definition of well-formed composite components.



#### Definition 4.9: Well-Formed Composite Component

Let  $k = (I_p, I_r, K_{sub}, Conn, Dele_p, Dele_r)$  be a composite component. Component  $k$  is **well-formed** iff the following conditions hold:

**WF<sub>1</sub>:** Let  $I'_r$  be any sub-component's *required interface*. Then, each method signature  $m' \in I'_r$  is linked either to a method signature of a sub-component's provided interface or – if there remain unlinked method signatures –  $I'_r$  is an element of the set of delegate connectors  $Dele_r$ . Therefore,  $I'_r$  has to be a valid refinement of the merge of all connected provided interfaces of sub-components and the component's required interface, and the following condition must be satisfied:

$$\forall k' \in K_{sub} : k'.I_r \prec \text{merge}(\{I'_p \mid (k'.I_r, I'_p)\} \cup \{I_r \mid k'.I_r \in Dele_r\}).$$

**WF<sub>2</sub>:** Each method signature in  $I_p$  is linked exactly to one method signature of a provided interface of a sub-component through a delegate connector. Therefore, the following condition must be satisfied:  $I_p \prec \text{merge}(Dele_p)$ .

**WF<sub>3</sub>:** All sub-components in  $K_{sub}$  are well-formed.

In line with component-based software engineering, well-formed composite components constitute (partial) correct-by-construction architectures. First, each well-formed composite component is self-contained, such that all requirements of required interfaces of sub-components are fulfilled. Furthermore, with component interface refinement (see [Definition 4.6](#)), we provide a checkable condition that prohibits invalid connections between interfaces during construction. Second, behavior of atomic components is implemented following the correct-by-construction approach (see [Definition 4.7](#)), which guarantees that the implementation of provided functionality complies with its specification. Finally, well-formed composite components are translated to source code, which we illustrate in the next subsection using JAVA as example. As our component model is simple on purpose, it is straightforward to manually verify that our code generation procedure leads to correct-by-construction source code.

#### 4.2.4. Excursion: ARCHICORC Code Generation in JAVA

So far, we introduced the high-level structure of the ARCHICORC component model and explained how CORC programs are bundled together. Although we will discuss the role of ARCHICORC in our verification and validation pipeline in the next section, the initial intention was to make correct-by-construction components available for larger software projects. In particular, after successful construction, well-formed ARCHICORC components can be translated to JAVA implementations with JML [[Leavens et al. 2006](#)] contracts. Then, the generated code can be imported as every other JAVA package. For the sake of completeness, we explain the JAVA/JML code generation process in more detail.

Considering an ARCHICORC component model subject to code generation, only provided methods of the top-level composite component are visible and accessible from outside the architecture. At the same time, the goal is to retain correctness during code generation. As CORC was de-

veloped following the principles of JAVA already, there exists a trivial translation from CORC programs to JAVA implementations [Runge et al. 2019a]. In Table 4.1, we illustrate the correspondence of the constructs of ARCHICORC components and generated JAVA implementations, which we discuss in more detail afterwards. In general, all generated interfaces and classes will be part of the same JAVA package. In particular, provided interfaces will simply be translated to regular JAVA interfaces with specified method signatures, where fields are replaced by respective `get` and `set` methods. The actual components are translated to JAVA classes that implement the corresponding provided interface and implementations for required interfaces will be propagated accordingly using helper classes. In particular, top-level components will be publicly accessible, whereas components on a lower hierarchy are only visible internally.

Formal Construct	Transformation
① For all required / provided interfaces $I = (M, F)$	<pre> <b>interface</b> <math>I</math> {   <math>\forall (\{\phi\} m(T_1 a_1, \dots, T_n a_n) \rightarrow T_{res} r\{\psi\}) \in M :</math>     <i>//@ requires <math>\phi</math>;</i>     <i>//@ ensures <math>\psi</math>;</i>     <b>public</b> <math>T_{res} m(T_1 a_1, \dots, T_n a_n);</math>    <math>\forall f \in F_p</math> with type <math>T</math>:     <i>//@ ensures \result == f;</i>     <b>public</b> <math>T</math> <code>get_f()</code>;      <i>// Only for provided interfaces:</i>     <i>//@ ensures <math>f == f'</math>;</i>     <b>public void</b> <code>set_f(<math>T f'</math>)</code>; } </pre>
② Atomic component $K \in K_{atom}$ with provided interface $I_p = (M_p, F_p)$ and re- quired interface $I_r = (M_r, F_r)$ .	<pre> <b>class</b> <math>K</math> <b>implements</b> <math>I_p</math> {   <math>\forall f \in F_p</math> with type <math>T</math>:     <b>private</b> <math>T f;</math>     <i>//@ ensures \result == f;</i>     <b>public</b> <math>T</math> <code>get_f()</code> { <b>return</b> <math>f</math>; }     <i>//@ ensures <math>f == f'</math>;</i>     <b>public void</b> <code>set_f(<math>T f'</math>)</code> { <math>f = f'</math>; }   <b>If</b> <math>I_r</math> is non-empty:     <b>public</b> <code>ReqAdapter<math>K</math></code> <code>req =</code>       <b>new</b> <code>ReqAdapter<math>K</math></code>(); ④     <i>// Method implementations ... ③</i> } </pre>

③ Method implementation for atomic component  $K = (I_p, I_r, \text{Impl})$  with  $I_p = (M_p, F_p)$  and  $I_r = (M_r, F_r)$ .

```
...
 $\forall(\{\phi\}m(T_1 a_1, \dots, T_n a_n) \rightarrow T_{res} r\{\psi\}) \in M_p:$ 
    //@ requires
    //@  $\phi[f \in F_r \mapsto req_K.get\_f];$ 
    //@ ensures
    //@  $\psi[f \in F_r \mapsto req_K.get\_f];$ 
    public Tres m(T1 a1, ..., Tn an) {
        if Tres is not void:
            return Implm(a1, ..., an); //CorC impl.
        otherwise:
            Implm(a1, ..., an);
    }
...

```

④ Adapter for required interface  $I_r$  of component  $K \in K_{atom} \cup K_{comp}$ , which is a sub-component of a composite component  $K^* = (I_p^*, I_r^*, K_{sub}^*, Conn, Dele_{req}, Dele_{prov})$ . We only depict the adapter implementation for internal connection between interfaces of sub-components. The adapter implementation for delegate connectors of required interfaces is analogous. The only difference is that instead of considering pairs from set Conn, we consider interface  $I_r^*$  and set  $Dele_{req}$ , such that the add method also accepts implementations of  $I_r^*$ .

```
class ReqAdapterK implements Ir {
     $\forall(I_p^i, K.I_r) \in Conn:$ 
        private Ipi provideri;
        public add(Ipi elem) {
            provideri = elem;
        }
         $\forall f \in F_r$  with type T and Ipi.f matches f:
            //@ ensures \result == f;
            public T get_f() {
                return provideri.get_f();
            }
         $\forall(\{\phi\}m(T_1 a_1, \dots, T_n a_n) \rightarrow T_{res} r\{\psi\}) \in M_r$ 
        and Ipi.m matches m:
            //@ requires  $\phi;$ 
            //@ ensures  $\psi;$ 
            public Tres m(T1 a1, ..., Tn an) {
                return provideri.m(a1, ..., an);
            }
    }
}

```

⑤ Composite component  $K \in K_{\text{comp}}$  with  $K = (I_p, I_r, K_{\text{sub}}, \text{Conn}, \text{Dele}_{\text{req}}, \text{Dele}_{\text{prov}})$ . If  $K$  is the top-level component, then  $\text{modifier} = \text{public}$ . Otherwise,  $\text{modifier}$  is removed.

```

modifier class K implements Ip {
   $\forall K_{\text{sub}}^i \in K_{\text{sub}}$ :
    private Ksubi subi = new Ksubi();
    // Getter / Setter:
     $\forall f \in F_p$  with type  $T$  and  $K_{\text{sub}}^i.I_p \in \text{Dele}_{\text{prov}}$ 
    and  $f$  matches  $K_{\text{sub}}^i.I_p.f$ :
      //@ ensures \result == subi.get_f();
      public T get_f() {return subi.get_f();}
      //@ ensures subi.get_f() == f';
      public void set_f(T f') {subi.set_f(f');}
    If Ir is non-empty:
      public ReqAdapterK req = new ReqAdapterK();
      // Constructor ... ⑥
      // Method implementations ... ⑦
}

```

⑥ Constructor for composite component  $K = (I_p, I_r, K_{\text{sub}}, \text{Conn}, \text{Dele}_{\text{req}}, \text{Dele}_{\text{prov}})$  with required interface  $I_r = (M_r, F_r)$ .

```

// Argument only if K is top-level component
public K(Ir arg) {
  req.add(arg);
   $\forall K_{\text{sub}}^i \in K_{\text{sub}}$  with  $K_{\text{sub}}^i.I_r \in \text{Dele}_{\text{req}}$ :
    subi.req.add(req);
   $\forall K_{\text{sub}}^i, K_{\text{sub}}^j \in K_{\text{sub}}$  with  $(K_{\text{sub}}^i.I_p, K_{\text{sub}}^j.I_r) \in \text{Conn}$ :
    subj.req.add(subi);
}

```

⑦ Method implementation for composite component  $K = (I_p, I_r, K_{\text{sub}}, \text{Conn}, \text{Dele}_{\text{req}}, \text{Dele}_{\text{prov}})$  with  $I_p = (M_p, F_p)$  and  $I_r = (M_r, F_r)$ .

```

 $\forall (\{\phi\} m(T_1 a_1, \dots, T_n a_n) \rightarrow T_{\text{res}} r\{\psi\}) \in M_p$ :
  //@ requires
  //@    $\phi[f \in F_r \mapsto \text{req}_K.\text{get}_f]$ ;
  //@ ensures
  //@    $\psi[f \in F_r \mapsto \text{req}_K.\text{get}_f]$ ;
  public Tres m(T1 a1, ..., Tn an) {
    There exists  $K_{\text{sub}}^i \in K_{\text{sub}}$ , such that
     $m$  matches  $K_{\text{sub}}^i.I_p.m$  with  $K_{\text{sub}}^i.I_p \in \text{Dele}_{\text{prov}}$ :
    return subi.m(a1, ..., an);
  }

```

Table 4.1.: Excerpt of code Translation from ARCHICORC components to correct-by-construction JAVA code [Knüppel et al. 2020b].

In the following, we briefly elaborate on each of the seven sections in the code generation process illustrated above.

1. Initially, all required and provided component interfaces are translated to regular JAVA interfaces including JML contracts for all method signatures. For fields, corresponding methods are added. Required interfaces only add get-methods, whereas provided interfaces also add set-methods.
2. Atomic components are translated to regular JAVA classes with package visibility implementing their provided interface. For non-empty required interfaces, a specialized helper class with the name `ReqAdapter` is instantiated that is used for resolving access to methods from the required interface (see 4).
3. Provided methods of atomic components are implemented by CoRC programs, which may use `ReqAdapter` to invoke *required* methods or simply invoke any of the provided methods.
4. The *adapter* class is a helper class for components that is used for resolving access to methods from the required interface. The class is needed, as our formalism allows to have a n-to-1 mapping for connections between provided and required interfaces. That is, we allow that numerous provided interfaces implemented by other components cover only parts of the required interface, but demand that merging all these provided interfaces satisfies the required interface (see [Definition 4.9](#)). In particular, there exist two versions of this class. The first version (which is illustrated in [Table 4.1](#)) connects the required interface of a component with all *connected* provided interfaces on the same level, and resolves the respective method calls. The second version considers delegate connectors (i.e., connections between two required interfaces), and is constructed analogously.
5. A composite component implements its provided interface and instantiates all contained sub-components. If it is the top-level component, the class becomes publicly available. Otherwise, the class is only visible inside the package.
6. The constructor of a composite component aims at establishing all relevant connections. First, the required interface of the composite component is delegated to the corresponding required interfaces of sub-components. Second, the constructor connects provided interfaces with the right adapter for required interfaces of sub-components. If the composite component is the top-level component and its required interface is non-empty, the respective JAVA interface is added as an argument, and an implementation must be provided by end-users.
7. Instead of implementing provided methods by CoRC programs, the matching method from the respective sub-component is resolved and invoked using the set of provided delegate connectors. If the result type of the method is non-void, the result value is returned. Otherwise, nothing is returned.

**Example 4.5.** In our running example, we modeled the thermostat as a composite component containing atomic components `Heater` and `On`. In [Listing 4.8](#), we show the generated code based on the thermostat composite component. Although we did not consider a required interface for the thermostat component before, we added interface `ISensorAndParametersR`, which gives access to the sensed temperature `x` and constants `u1`, and `K`.

The constructor ensures that both objects `heater` and `on` are rightfully connected, and that object `on` has access to the user-provided implementation for `ISensorAndParametersR`. Furthermore, class

---

```

1 package ThermostatExample;
2 public class Thermostat implements IHeaterP {
3     public ReqAdapterThermostat req = ReqAdapterThermostat();
4     private Heater heater = new Heater();
5     private On on = new On();
6
7     public Thermostat(ISensorAndParametersR arg) {
8         req.add(arg);
9         heater.req.add(on);
10        on.req.add(req);
11    }
12    //@ ensures h == heater.get_h();
13    public void set_h(double h) {
14        heater.set_h(h);
15    }
16    //@ ensures \result == heater.get_h();
17    public double get_h() {
18        return heater.get_h();
19    }
20    //@ requires  $\phi$ ;
21    //@ ensures  $\psi$ ;
22    public void ctrlStepHeater() {
23        heater.ctrlStepHeater(); //Correct-by-construction
24    }
25 }

```

---

Listing 4.8: A thermostat implementation generated by ARCHICORC.

Thermostat exposes the control method `ctrlStepHeater()` from the `IHeaterP` interface, as shown in [Line 22](#). Method `heater.ctrlStepHeater()` is the correct-by-construction implementation of the control method developed with CORC and – depending on the sensor input – modifies the heating value `heater.h`. After deployment, an end-user can interface with the thermostat class by providing an implementation for `ISensorAndParametersR` (i.e., the sensor component), invoking the control method, and finally propagating the value from variable `h` to the actuating component.

### 4.2.5. Discussion

We consider the ARCHICORC component model to be starting point for implementing *correct* and *reusable* libraries (or packages) following the correct-by-construction approach. In this section, we discuss current obstacles and limitations, how they can be overcome, and future directions.

#### Specifications Beyond Preconditions and Postconditions

Following Hoare-style reasoning [[C. A. R. Hoare 1981](#)] and design by contract [[Meyer 1992](#)], our formulation in the previous sections considers only simple contracts comprising one precon-

dition and one postcondition. In the last decade, more advanced specification concepts were introduced to make specifying software more practical and enjoyable, and sometimes to make verifying software automatically even feasible.

**Framing.** By introducing method calls in CORC programs (see [Section 2.1.3](#)), employed verifiers must know, which locations are possibly changed and which locations are guaranteed to remain unmodified. In the current version of ARCHICORC, this information is encoded as  $\wedge_i v_i = v_i^{\text{old}}$ , where  $v_i^{\text{old}}$  expressed the value of  $v_i$  before method execution. Instead, specification effort can be reduced by introducing additional syntactic sugar called framing condition [[Hatcliff et al. 2012](#)] or modifier clause [[Liskov et al. 1986](#); [Leino 1998](#)], such as the keyword **assignable** as used in JML [[Leavens et al. 2006](#)], to use a set-theoretic notion for modifiable locations.

**Multiple Specification Cases.** Modern specification languages support the definition of more than one contract for one method referred to as *specification cases*. For instance, JML provides the keyword also [[Chalin et al. 2005](#)] to connect multiple contracts. In essence, each specification case defines a different behavior with respect to a unique assumption. Again, separation of specification cases is mostly syntactic sugar, as shown by Chalin et al. [[2005](#)]. That is, given  $n$  specification cases with their unique precondition  $\phi_i$  and postcondition  $\psi_i$ , translating all  $n$  specification cases into a single contract leads to precondition  $\bigvee_i^n \phi_i$  and postcondition  $\bigwedge_i^n \phi_i^{\text{old}} \Rightarrow \psi_i$ . However, explicit syntax for specification cases certainly improves readability of specifications and will be addressed in the future.

**Class Invariants.** Besides method contracts, another kind of specification is represented by *class invariants* [[Leavens et al. 2007](#)], which allow to specify unchangeable conditions of class members with one-time effort (i.e., only once per class in object-oriented programs). Consequently, methods must implicitly conform to the class invariant before and after any method execution, while the alternative is to add the invariant's condition to each precondition and postcondition explicitly. Class invariants in CORC are represented by *global conditions* (see [Section 2.1.3](#)). In ARCHICORC, however, there is currently no concept for specifying invariants that must be respected by all methods of an interface. We aim at adding a special keyword to ARCHICORC's interface description language as part of future work. Global conditions of corresponding CORC programs then have to be implied by the invariant.

### Listkov-style Compatibility between Specification and Data Types

ARCHICORC aims at managing correct-by-construction library functions and therefore connects interface methods with CORC programs based on refinement. By relying on CORC, a crucial step is the last refinement (e.g., typically assignments), which involves a move from the specification language to the programming language. It is therefore important to think about how data types of the specification language are represented in the programming language, consequently leading to potential errors. For instance, natural numbers in specifications are often represented using an abstract infinite domain, whereas JAVA, for example, bounds natural numbers to a finite domain.

There are three possible solutions to this problem [[Beckert et al. 2005](#)]. First, if possible, we could change data types on the implementation level to exhibit the same semantics as pro-



vided by the specification language. Second, we may do the opposite by modeling data types of the specification language similar to their implementation-level representation. Third, we may think of a *controlled incorrectness* in our refinement rules, usually referred to as *retrenchment* [Banach et al. 1998; Beckert et al. 2005]. A retrenchment framework is able to prove the *absence* of cases that lead to such problems.

As ARCHICORC’s focus is primarily on component-based design for CORC programs and code generation, it shifts this problem to CORC itself. In particular, CORC does not violate the refinement from specification to implementation by integrating JML as specification language and by employing the deductive verifier KEY as part of its tool chain. Consequently, CORC applies the second solution by only allowing implementation-level data types in its specifications. However, as identified by Beckert et al. [2005], there are certain drawbacks of this solution. For instance, hiding of implementation details is reduced and not all implementation details are even known during the specification phase.

In summary, a promising direction is to study retrenchment in combination with the correctness-by-construction approach. Retrenchment promotes the idea of implementation hiding by focusing on more abstract specifications for refinement calculi beyond implementation-level data types. Moreover, with the rise of functional programming, developers of a future generation may think of types as mathematical objects with infinite domains rather than class-related objects as promoted by object-oriented programming. As mentioned before, lifting the data types of established programming languages to infinite domains is infeasible. Therefore, allowing infinite domains for data types of specification languages is best addressed by applying retrenchment.

### Floating-Point Support in Deductive Verification

CORC integrates the deductive program verifier KEY as back-end in its tool chain. A big challenge is to reason about floating-point arithmetic conforming to the IEEE 754 standard, which has only very limited support in the current release version of KEY. In contrast, there exist other deductive verifiers, such as FRAMA-C [Cuoq et al. 2012] for C and SPARK [Barnes 2012] for Ada, who support floating-point reasoning more properly. Unfortunately, ARCHICORC is built on top of CORC and also aims at bridging the gap between hybrid systems, which rely heavily on real arithmetic, and machine-executable code. Consequently, a large part of the generated and constructed CORC programs in the translation process (which we discuss in the next section) remains unverified or at least requires high manual verification effort.

While this is currently a rather unpleasant drawback of our tool chain, there exist very recent efforts to integrate proper floating-point reasoning into KEY [Abbasi et al. 2021]. Instead of adding new rules in KEY’s sequent calculus for directly handling floating-point arithmetic, SMT solvers with a theory of real arithmetic are integrated (i.e., similar to KEYMAERA X). Slight additions to JML, however, require CORC to potentially enhance its specification language as well. As ARCHICORC directly interfaces with CORC, proper floating-point reasoning becomes available as soon as CORC updates to a supporting KEY version.

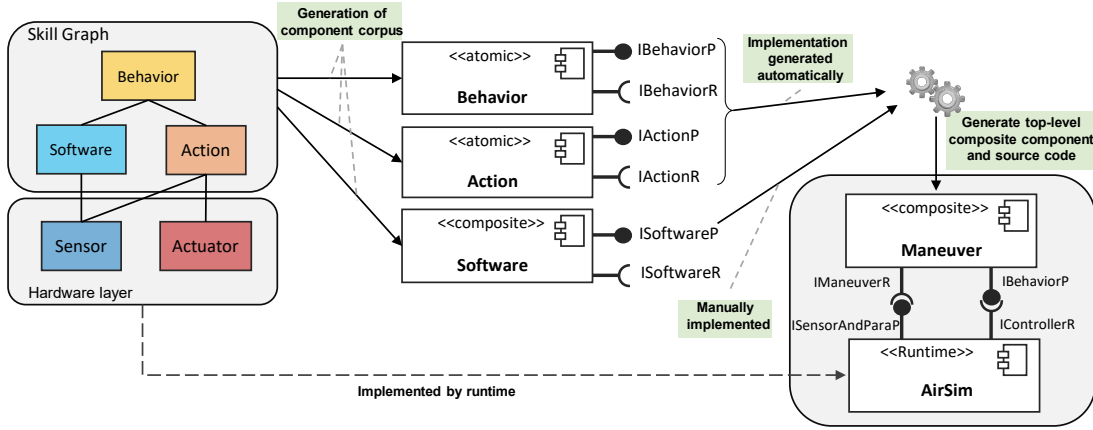


Figure 4.3.: Transformation of skill graphs to ARCHICorC components.

### 4.3. Tool Support for Virtual Validation of Skill Graphs

In the previous section, we introduced the ARCHICorC component model and hinted at the fact that skill graphs are translated to component-based architectures. The goal is to support incremental model and specification debugging (i.e., *is my modeled maneuver accurate enough for the real world?*), which is best addressed by virtual validation. At the same time, ARCHICorC incorporates deductive verification and correctness-by-construction on source-code level, which is another layer to reduce potential defects when making the leap from model to implementation. To aid developers in deriving *verifiable* implementations that conform to a model's needs, we provide tool support in SKEDITOR for implementing skill graphs with ARCHICorC components. In Figure 4.3, we give a simplified overview of the relationship between skill graphs and ARCHICorC components, which we briefly explain in the following.

After a verified skill graph is developed, both software and hybrid skills are translated to boilerplate ARCHICorC components. In particular, each component is equipped with a specialized requires and provided interface. Software skills are translated to composite components with their required and provided interface consisting of their input and output variables, respectively. Additionally, each such component provides method `void update(void)`, which has the purpose to (1) call the update method of dependent software skills and (2) update their respective variables. Internally, the update method is implemented manually following the guidelines of ARCHICorC. Hybrid skills are translated to atomic components fully automatically based on their controller implementation in SL, which we explain in more detail in Section 4.3.1. Next, the ARCHICorC components are connected following the dependency structure of the original skill graph. This results in a new top-level composite component that requires sensor values and parameters, and exposes the controlling function of the top-level behavioral skill. Finally, implementation of hardware skills (i.e., sensors and actuators) is given by a simulation platform supported by ARCHICorC, which must interface with the composite component to establish a closed-loop feedback control.

To transfer the model-level execution semantics established in the previous chapter (see Section 3.2.2) to the implementation level, each supported simulation platform is required to provide an instantiable template implementation of the control loop employed at run-time. As

this is a manual one-time effort per simulation platform and the template is instantiated for all maneuvers equally, we argue that the instantiation is correct-by-construction. As we primarily focus on time-triggered systems, we reserve the variables  $t$  and  $ep$  measured in milliseconds for clock time and maximum elapsed time between two sensor measurements, respectively. As  $ep$  is typically undefined in the modeling phase, it must be concretized in the respective template implementation of the control loop.

An example of such a simulation environment that we integrated is AirSim [Shah et al. 2018], which allows us to translate ARCHICORC components to C++ code and simulate the respective maneuvers in realistic 3D environments with highly accurate physics. In Section 4.3.1, we explain how skills are translated to ARCHICORC components and how source code is generated. In Section 4.3.2, we show how simulation platforms interface with our tool chain using AirSim as example.

#### 4.3.1. Automating ARCHICORC Component Generation

In the proposed development process, each skill in a skill graph is mapped to an ARCHICORC component. For software and hardware skills, each skill is mapped to an ARCHICORC composite component with one required and one provided interface that mirror the dependencies of the skill-graph model. For instance, consider a *perception skill* for identifying obstacles in front of a vehicle that gets as input sensor values from a Lidar sensor and outputs a list of obstacles. The required interface will request a Lidar sensor object provided by a Lidar sensor component, while the provided interface will provide the list of obstacles to connected, higher-level skills. In contrast to hybrid skills, which are only allowed to use real values, components based on software skills may communicate with arbitrary objects in their interfaces, as long as the respective classes are available. We exemplify the required and provided interface of such a software skill in the following.

---

<pre> <b>archicorc_interface</b> IPerceptionR {     // Lidar sensor     Lidar sensor_data; } </pre>	<pre> <b>archicorc_interface</b> IPerceptionP {     // List of objects     List&lt;ObstacleData&gt; obstacle_data;     //@ requires <math>\phi</math>;     //@ ensures <math>\psi</math>;     <b>void</b> updatePerception(<b>void</b>); } </pre>
---	---

---

As can be seen, each provided interface gets a unique *update method* for updating all provided variables in the current cycle. This method must be implemented internally by end-users using atomic components. Although we will not go into too much detail on how the update process works, as it is straightforward, there are two implicit aspects that arise during the code generation process, but are indeed hidden from end-users. First, invoking the update method of a skill will also first automatically invoke all update methods of dependent software skills on lower levels. This ensures that required variables are always up-to-date. Second, invoking the same update method more than once during the same cycle will not have any additional effect, which saves computational power. As ARCHICORC promotes the correctness-by-construction approach, we hope that implementation defects can be reduced.

For hybrid skills, each skill is mapped to an ARCHICORC atomic component with one required and one provided interface, where the respective code of provided methods is mostly generated automatically. The reason is that hybrid skills are based on hybrid mode automata and SL programs, which allows us to translate this behavior to general-purpose programming languages. In fact, as SL is syntactically and semantically close to hybrid programs, our translation process is inspired by KEYMAERA X's own code generation process [A. Müller et al. 2018b].

Each component resulting from a hybrid skill provides a *control method* that modifies the current state once according to the underlying hybrid mode automaton. Control methods of dependent hybrid skills are added to the required interface. Importantly, we allow modelers to abstract away certain parts in their formalization using nondeterministic assignment, which must be resolved during implementation. We solve this by adding additional methods to the provided interface attached to the respective component that must be implemented by hand. The following interfaces for a behavioral skill exemplify this solution.

---

<pre> <b>archicorc_interface</b> IBehaviorR {     // Parameters     <b>double</b> <math>p_1, \dots</math>;     // Input variables     <b>double</b> <math>i_1, \dots</math>;      // Dependent hybrid skills     //@ requires <math>\psi_{act}^1</math>;     //@ ensures <math>\psi_{act}^1</math>;     <b>void</b> ctrlStepAction<sub>1</sub>(<b>void</b>);     // Output from Action     <b>double</b> <math>o_i \in \{o_1, \dots\}</math>;     // ... } </pre>	<pre> <b>archicorc_interface</b> IBehaviorP {     // Output variables     <b>double</b> <math>o_1, \dots</math>;     // State ID     <b>int</b> state_id;      //@ requires <math>\phi_{ctrl}</math>;     //@ ensures <math>\psi_{ctrl}</math>;     <b>void</b> ctrlStepBehavior(<b>void</b>);      // Resolve nondet. assignment     //@ requires <math>\phi_{ndet}^1</math>;     //@ ensures <math>\psi_{ndet}^1</math>;     <b>double</b> a<sub>1</sub>();     // ...      <b>bool</b> monitorSatisfied(State prior); } </pre>
---	---

---

As illustrated above, key element of the provided interface is the `ctrlStep` method that advances the state each cycle. Automated code generated for the `ctrlStep` methods resembles the underlying structure of the hybrid mode automaton. An important question is how the specification (i.e., precondition  $\phi_{ctrl}$  and postcondition  $\psi_{ctrl}$ ) is obtained. We solve this issue by translating the corresponding hybrid mode automaton to a *logical formula*, which we then use as postcondition, while using the skill's assumption and environmental conditions as precondition.<sup>4</sup> Finally, we add a method to the top-level component for *monitoring* whether any control actions violate the original (and verified) controller model. Input argument `State` is a simple struc-

---

<sup>4</sup>Translating the hybrid mode automaton to a logical formula is largely inspired by the translation process of hybrid programs to  $d\mathcal{L}$  [A. Müller et al. 2018b; Mitsch et al. 2016]. We omit further details, as the procedure is only of technical nature.

ture used as shorthand for the collection of input and output variables. We discuss the generation of monitor code at the end of this section.

In the following, we explain how the main control function – `ctrlStep` – is generated from the initial discrete control structure represented by hybrid mode automata. Although there exists a directly corresponding CorC implementation of the `ctrlStep` method, we discuss the translation only using plain C++ for the purpose of readability. We start with the following structure and explain the translation informally for each aspect (i.e., modes and discrete programs) of a hybrid mode automaton. Methods prefixed with two underscores are internal helper methods.

---

```
void ctrlStep(void) {
    State result = __updateAndGetState();
    // [...generated controller logic...]
    __updateState(result);
}
```

---

Internally, we use a unique `State` object to bundle all input and output variables together. The two helper methods illustrated above are essentially wrappers for all get and set methods, and are used to initialize the temporary state object and to update the output variables of the respective component. Additionally, method `__updateAndGetState()` initially invokes all update methods of dependent software skills, such that each cycle necessary output variables of software skill are updated. In the following, we explain how code for the `ctrlStep` method is generated.

**Modes.** Similar to the previous chapter, each mode gets a unique mode identifier `state_id`  $\in \mathbb{N} \cup \{0\}$  and `ctrlStep` starts with transitioning to the correct state using conditionals:

---

```
// Transitions
if(current.state_id == 0) {
    if(__condid=00)
        result.state_id = ...;
    else if(__condid=01)
        result.state_id = ...;
    else if ...
}
else if(current.state_id == 1) {
    ...
}
else if ...
```

---

Symbol `__condidi` returns the  $i$ th guard of the respective transition from the current mode to another mode. For instance, in our thermostat example, modes `Off` and `On` are associated with `state_id` 0 and 1, respectively. If temperature  $x$  almost reaches lower limit 1, the thermostat system transitions to mode `On`. In the translation process, `__cond00` equals  $x \leq 1 + \delta_l$ , with  $\delta_l \geq 0$ . In contrast to KEYMAERA X [A. Müller et al. 2018b], we eliminated arbitrary nondeterministic choice in our models, which makes the generation process more straightforward.

Afterwards, each mode itself is then translated surrounded by conditionals based on the current state of `result.state_id` similar to the structure above. Next, we explain how each mode including deterministic and nondeterministic statements, terms, formulas, and submode abstractions is translated.

**Deterministic Statements.** Discrete assignments and `if – else` statements are directly translated to C++ code. That is,  $x := \theta$  is translated to `result.x = trans_expr( $\theta$ )`; and `if( $H$ ){ $A$ }else{ $B$ }` is directly translated to:

---

```
if(trans_formula(H)) {
    trans_stmt(A);
} else {
    trans_stmt(B);
}
```

---

The three translation functions prefixed with `trans_` are recursively resolved until they result in syntactically-correct C++ code. A particularity is represented by `assume` and `assert` statements. Both statements leave the state unchanged but will report on violations during execution.

**Terms and Formulas.** One problem that arises from translating models based on  $d\mathcal{L}$  (or most other abstract representations of cyber-physical systems) to machine-executable code is the handling of real arithmetic. Floating-point imprecision of real-valued variables can lead to violations not based on the control logic, but solely based on the underlying architecture. Ensuring safety margins and guaranteeing a sound representation in machine-executable code is therefore of paramount importance for real production code, as pursued by the VeriPhy project [Bohrer et al. 2018] for hybrid programs. As the main goal of this thesis is restricted to simulation and functional correctness only, we provide only an unsound translation. Nevertheless, we translate all parameters and variables used to a floating-point data type with the highest precision of the underlying architecture. For C++, this is achieved by using the `long double` primitive data type.<sup>5</sup> When variables or parameters are accessed in terms or formulas, we compile them by either using the current state (i.e., `result.x` for accessing variable  $x$ ) or the globally accessible parameter structure (i.e., `parameters.A` for accessing constant  $A$ ).

**Nondeterministic Assignment.** Nondeterministic assignment is often used to introduce variability into the controller code. That is, for a given variable, the controller should be safe regarding a variety of values instead of only a concrete one. As mentioned before, we generate specific methods as part of the interfaces for these occurrences. We assume that nondeterministic assignments are typically paired with `assume`  $H$  statements, which may serve as a possible postcondition for these assignments. For such a given pair `havoc x; assume H`, we provide the following translation, where  $\psi_{ndet}^x \equiv H$ .

---

<sup>5</sup>If supported by the architecture and compiler, `long double` will match a 128-bit floating-point type. Otherwise, at least a 64-bit floating-point type is guaranteed.

---

```

/*@ requires  $\phi_{ndet}^x$ ;
   //@ ensures  $\psi_{ndet}^x$ ;
   long double x() { ... } /*Implement nondeterministic assignment */

void ctrlStep(void current) {
    //...
    result.x = this->x();
    //...
}

```

---

If possible, the helper function  $x()$  is constructable employing the correctness-by-construction approach and CORC or verifiable using post-hoc techniques [Ahrendt et al. 2016]. For the precondition  $\phi_{ndet}^x$ , we only insert facts about the current parameters, as given by the skill-graph model. In the future, more sophisticated analyses based on weakest precondition are possible to derive a more precise precondition.

Another abstraction mechanism that we introduced in the previous chapter were Hoare triple specifications. Here, precondition and postcondition are explicitly given and translated to a valid specification as-is. In contrast to nondeterministic assignment, we assume that Hoare triples play a larger role in the modeling phase and ideally are resolved before an machine-executable implementation is derived. Therefore, we omitted the generation of dedicated methods for this abstraction mechanism.

**Submode Abstraction.** Recall that submodes in modes are used for activating the hybrid mode automaton of a lower level skill (see Chapter 3). In our generated controller code, this means that the `ctrlStep` of another component is invoked and must consequently be accessible through the required interface:

---

```

void ctrlStep(void) {
    //...
    __setState(&OtherComponent, &result);
    OtherComponent->ctrlStep();
    result = __getState(&OtherComponent, &result);
    //...
}

```

---

Here, `OtherComponent` is an implementation of an interface that provides the method `ctrlStep` as well. To avoid name clashes, the `ctrlStep` is typically suffixed with a unique identifier. Helper methods are again used to update respective variables of the dependent component and to transfer any state changes to the upper-level component.

**Generating Monitor Code.** As apparent in our verification and validation pipeline, we make the leap from a verified controller model to an *untrusted* controller implementation in C++. The reason is that (1) nondeterminism must be resolved, but how it must be resolved is free to the developer, and (2) we (at least) provide means for verifying the correctness of the implementation,



but due to practicality, we do not enforce it. In particular, conditions used in the  $d\mathcal{L}$  model, such as the initial safe prestate or the safety guarantee, are eliminated in the process of deriving an implementation. To still be able to assess whether our controller behaves as intended, we additionally generate monitor code for our top-level controller in a dedicated method with signature `bool monitorSatisfied(State prior)`. Eventually, the monitor code can be used throughout runtime to report condition violations of the controller (i.e., deviations of controller model and implementation) that would otherwise be unnoticed.<sup>6</sup>

The generated code compares the current state of the controller with the previous state, which we explicitly provide as argument (the current state is obtained from the atomic component itself). If the current state is not a valid poststate of the given prestate according to our modeled controller, the monitor is not satisfied. We exploit MODELPLEX [Mitsch et al. 2016] in our process that, given a  $d\mathcal{L}$  model, generates monitor code automatically. Besides generating monitor code for the controller, it is also possible to generate monitor code based on the modeled dynamics. This way, it is possible to assess whether the real world behaves according to the assumed world. In our tool chain, we will only generate code that monitors the controller.

**Example 4.6.** *We revisit a simplified version of the thermostat example. Translating the controller code to  $d\mathcal{L}$  may result in the following logical formula:*

---

```
state_id = 0 & x >= u() & (
  x >= l() & hpost = h & statepost = 1 & xpost = x
)
| state_id = 1 & x <= l() & (
  hpost >= u() & x <= u() & statepost = 0 & xpost = x
)
)
```

---

Depending on `state_id`, the formula represents valid conditions for changes resulting from executing the controller. For instance, the controller itself does not modify temperature `x` in any state, which is why before and after executing the controller, `x` must remain unchanged (highlighted). Translating the logical condition to C++ code results in the following implementation:

---

```
bool monitorSatisfied(State prior) {
  return (prior.state_id == 0.0L && prior.x >= this.params->u
    && prior.x >= this.params->l && this.h == prior.h
    && this.state_id == 1.0L && this.x == prior.x)
  || (prior.state_id == 1.0L && prior.x <= this.params->l
    && this.h >= params->u && prior.x <= this.params->u
    && this.state_id == 0.0L && this.x == this.x);
}
```

---

<sup>6</sup>As presented by Bohrer et al. [2018], it is common practice to also provide fallback maneuvers in case of a severe condition violation (e.g., an emergency brake), which is why monitoring is indispensable during execution.

### 4.3.2. Example: Interfacing with AirSim

To conclude this section, we explain how the generated ARCHICorC components from skill-graph models can be used with AirSim. Essentially, the generated code implements a step in the control loop accessible by the top-level behavior skill of the skill graph. Based on the current state given by the sensors, the control step is executed to compute the post state. Afterwards, essential variables of the post state are propagated to the actuating components. In our tool chain, we automated this process for AirSim's car simulation. To illustrate how the generated code may interface with a simulation platform such as AirSim, we present a stripped-down version for the *instantiated* main loop of a *lane keeping assistance* maneuver in Listing 4.9.

---

```

1 ParametersLKA params;
2 params.ep = EP;
3 //...
4 msr::airlib::CarRpcLibClient client = ...; // Init.
5 LaneKeepAssistance assist = new LaneKeepAssistance(&params);
6 //...
7 while (loop(chrono::duration_cast<chrono::milliseconds>(end - begin))) {
8     begin = std::chrono::steady_clock::now();
9     // ① Sense (generated by ArchiCorC)
10    StateLKA current = __getCurrentState(); //get position,
11                                         //velocity, etc. from simulation
12
13    // ② Control (generated by ArchiCorC)
14    __setState(&assist, &current);
15    assist->ctrlStepLKA();
16    StateLKA post = __getPostState(&assist);
17
18    // ③ Monitor (generated by ArchiCorC)
19    if(!assist->monitorSatisfiedLKA(current)) {
20        // Monitor violated -> report!
21    }
22
23    // ④ Actuate (manually implemented)
24    setThrottle(&client, post.a);
25    setBrake(&client, post.b);
26    setSteering(&client, post.w);
27
28    end = std::chrono::steady_clock::now();
29 }

```

---

Listing 4.9: C++ implementation of closed-loop control for a lane keeping assistance using AirSim.

Object `LaneKeepAssistance` represents the top-level behavior skill and provides method `ctrlStepLKA`. The `while`-loop represents the closed-loop feedback control. First, the updated state (e.g., car's position and velocity) is requested from the runtime ①. This is where sensor com-

ponents, which are implemented for each simulation platform differently, interface with the generated controller. Second, the control method `ctrlStepLKA` computes the next state based on the current one ②. Third, violation of monitored conditions is checked ③, which may either be reported or even trigger fallback behavior (e.g., emergency brake). Finally, computed values of control variables are propagated to the actuating components ④. Most importantly, all other implementation details are hidden from the runtime environment, allowing the seamless integration of our approach into other simulation environments.

## 4.4. Evaluation

The above sections and the previous chapter raised some questions that we try to investigate by empirical experiments and a qualitative assessment. In particular, we aim at discussing the following three research questions.

**RQ-1:** *To what extent do we rely on nondeterministic assignments in practice?*

**RQ-2:** *To what degree do correctness guarantees hold at the level of implementation?*

**RQ-3:** *To what extent is validation of skill graphs insightful with respect to an incremental development process?*

With **RQ-1**, we aim at investigating the portion of nondeterministic assignments in our case studies. Several modes may use nondeterministic assignments, which is sufficient (and even convenient) for the verification phase, but must eventually be resolved at the implementation level. Although we only consider a small number of case studies, the number of nondeterministic assignments can still give an impression on additional effort that must be spent during the implementation phase. Furthermore, nondeterminism exhibits an additional source of danger; while deterministic assignments are translated automatically to source code, nondeterministic assignments must be implemented manually and are only *assumed* during the verification phase. This is also the reason, why we directly integrated the correctness-by-construction approach in the architectural layer of our pipeline.

With **RQ-2**, we investigate the *feasibility* of our verification and validation pipeline. In particular, we evaluate whether safety guarantees can be transferred from the verification model to the execution model following the guidelines of ARCHICORC. Finally, with **RQ-3**, we discuss our experiences with the proposed development process.

We characterize our non-trivial case study of a road vehicle and the evaluated subject maneuvers in [Section 4.4.1](#). In [Section 4.4.2](#), we present results and discuss insights on the questions asked before. Finally, we discuss limitations and possible threats to validity in [Section 4.4.3](#).

### 4.4.1. Case Studies and Setup

Although the concept of skill graphs is applicable to a diverse set of cyber-physical systems, we aim at evaluating our pipeline in the context of automated driving similar to the conducted case study in [Section 3.5](#). In particular, we employ AirSim [[Shah et al. 2018](#)] as simulation environment to validate to what extent correctness guarantees of the skill-graph models can be transferred to

implementation level. For automatic driving, the final goal is indeed to develop skill graphs for each maneuver from a catalog of basic driving maneuvers that can be safely applied in road traffic.<sup>7</sup> The purpose of this evaluation, however, is a *proof of concept* of our verification and validation pipeline.

Our verification and validation pipeline proposes two novel modeling concepts that did not exist before, namely skill graphs presented in the previous chapter and ARCHICORC presented in this chapter. Consequently, we cannot expect to study real-world driving maneuvers and transform them into our approach error-free and without considerable effort. Instead, we created all case studies from scratch, which may serve as a baseline for further empirical assessments in the future.<sup>8</sup> The rationale behind creating skill graphs completely from scratch and translating them to machine-executable code is twofold. First, emphasis is on coming up with a model design that follows the language's intention. We believe that this way, modeled maneuvers display a higher quality in terms of design. Second, developing all case studies from scratch simultaneously allows us to think about how to reuse implemented skills across maneuvers and how to decrease verification effort upfront.

In total, we created five maneuvers including validation scenarios in AirSim to demonstrate the applicability of our verification and validation pipeline. All studied maneuvers are modeled as skill graphs, verified, and eventually implemented in C++ supported by ARCHICORC. As ARCHICORC is limited with respect to floating-point reasoning (see Section 4.2.5), parts of the implementation remain unverified. However, such parts are already in a form that allows to verify them when reasoning with floating-point arithmetic becomes available. A short description of the maneuvers and their safety requirements is given in the following.

**Explore World (Vehicle version).** This maneuver is similar to the *Explore World* maneuver described in Section 3.1.2 for the robotic domain. The goal is still to (randomly) explore an area without colliding with any object. The difference is that this maneuver is now applied to a vehicle with different kinematics preventing rotating on the spot. Instead, the turning circle must be considered explicitly. Safety goals are (1) respecting a maximum velocity ( $v \leq v_{max}$ ), and (2) avoid collision.

**Safe Halt.** The goal of this maneuver is to drive forward in a straight line and come to a halt if needed (i.e., either in front of an obstacle or a particular point) without collision or overstepping. Safety goals are again (1) respecting a maximum velocity ( $v \leq v_{max}$ ), and (2) avoid collision.

**Lane Keeping Assistance.** This maneuver captures the lateral aspects of following a lane, where it is not allowed to deviate too far from the lane's center point. For this, the vehicle must drive on a lane with perceivable solid lane markings. Safety goal is to respect a maximum lane deviation ( $y \leq y_{max}$ ).

**Adaptive Cruise Control (Unoptimized and Optimized).** This maneuver captures the longitudinal aspects of following a car while keeping a safe distance. We further split the *adaptive cruise control* maneuver into two versions, an optimized version and an unoptimized version. The optimized version resembles a more sophisticated controller on the modeled level without nonde-

<sup>7</sup>For example, a catalog of basic driving maneuvers as necessary for road traffic is given by Bergmiller [2015].

<sup>8</sup>Although all case studies were created from scratch, the majority was inspired either by existing concepts of skill graphs including an informal specification, or by some of the KEYMAERA X projects found online (<https://github.com/LS-Lab/KeymaeraX-projects>).

terministic assignments, whereas the unoptimized version is simpler and leaves more room for the implementation. Safety goal is to avoid collision with the leading vehicle.

As our evaluation aims at investigating whether our concept is feasible, we only built simple scenarios to test each driving maneuver in isolation. For the *Explore World* maneuver, we created a closed world with numerous static obstacles. For the *Safe Halt* maneuver, we used the same world, but placed the vehicle in front of an obstacle in a straight line. For the lane keeping assistance, we modeled the street and encoded the ground truth of the lane markings to eliminate sensor uncertainty. For the adaptive cruise control, we added a second, leading vehicle in front of our vehicle that drives in a straight line.

#### 4.4.2. Results and Insights

In the following, we share results obtained with our case studies. In particular, we are interested in usefulness and feasibility of our proposed approach. First, we investigate to what degree nondeterminism plays a role in our case studies. Next, we investigate whether safety guarantees from verified skill-graph models can be transferred to the implementation level. Finally, we discuss our experiences. The concrete models are contained as examples in SKEDITOR.<sup>9</sup>

##### RQ-1: Nondeterminism

In Figure 4.4, we illustrate the number of top-level modes (i.e., modes of the behavioral skill) and the number of nondeterministic assignments. Noteworthy, all Hoare triple specifications used during the modeling phase were eliminated beforehand (i.e., were only used for incremental development of the skill-graph models themselves). We discover that the *Optimized Adaptive Cruise Control* defines five modes, whereas all other case studies define only three modes. We additionally observe that the majority of case studies make use of nondeterministic assignment, with the *Explore World* maneuver being the front runner with five nondeterministic assignments. The two exceptions, the *Optimized Adaptive Cruise Control* and *Safe Halt*, use only discrete assignments.

The implementation effort for the nondeterministic assignments may differ greatly. For instance, assignments for acceleration  $a$  are often written in the models as `havoc a; assume  $-B \leq a \leq A$` , which leaves more room for the concrete implementation. For fast results, it is therefore tempting to just assign a constant value to  $a$  (e.g.,  $a := A$ ), which, however, often leads to non-optimal driving behavior. In the initial version of the *Safe Halt* maneuver, we had a similar nondeterministic assignment `havoc a; assume  $0 < a \leq A$` , which we replaced by the discrete assignment  $a := A$ .

Based on our results, we conclude that the correctness-by-construction approach is indeed a valuable paradigm to bridge the gap between nondeterminism and safe implementation. In particular, nondeterministic assignments are essentially tiny programs, where the postcondition can be identified either manually or automatically by a light-weight analysis.

<sup>9</sup><https://github.com/AlexanderKnueppel/Skeditor>

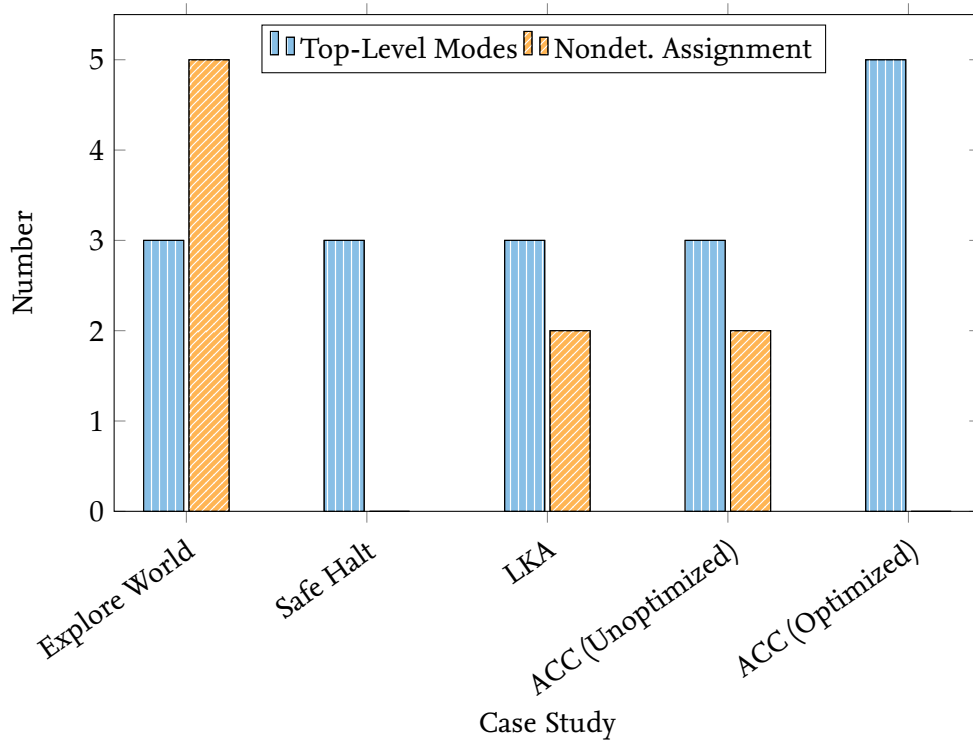


Figure 4.4.: Number of top-level modes (behavior) and nondeterministic assignments per case study.

## RQ-2: Validation and Safety Violations

In the following, we investigate the feasibility of our approach by simulating and validating the generated implementations of each case study in AirSim. As mentioned before, each maneuver was successfully verified, which leads to the question whether correctness guarantees transfer to the implementation level.

In [Table 4.2](#), we summarize results from the performed experiments. In particular, for the fail statistics, we report three values. First, column *passable* reports on the percentage of monitor violations that we consider as acceptable. These stem from violations of nondeterministic assignments or other issues in the initial state, but do not violate the safety invariants. Second, column *severe* reports on the percentage of violations of the safety invariants. Finally, *failure rate* is the time spent in monitor-violating states with respect to the simulation time.

We observe that none of the five maneuvers violated any of the safety invariants during their simulation, which is why all reported violations are considered as *passable*. Furthermore, the failure rates are all low. However, we identified that oftentimes the initial state (i.e., a halted state with zero velocity for all five case studies) violates the monitor condition, while during movement the failure rate converges towards zero. Other times, the failure rate increases for a short amount of time. We assume that the conversion of arithmetic reals to floating-point precision results in sporadic problems.

Maneuver	Safety Goal	Fail Statistics			Sim. Time
		Passable	Severe	Failure Rate ( $\emptyset$ )	
Explore World	$v \leq v_{max}$ ; no collision	100%	0%	1.07%	5m
Safe Halt	$v \leq v_{max}$ ; no collision	100%	0%	3.03%	ca. 6-12s.
LKA	lane deviation ( $y \leq y_{max}$ )	100%	0%	1.63%	30s
ACC (Unoptimized)	no collision	100%	0%	2.72%	30s
ACC (Optimized)	no collision	100%	0%	2.36%	30s

Table 4.2.: Simulation results.

### RQ-3: Experiences

In the following, we illustrate the obtained results and our gained experiences following the proposed development and simulation process using the *Optimized Adaptive Cruise Control* and *Safe Halt* case studies as two representative examples. In Figure 4.5, we depict the position, velocity, acceleration, and failure rate over time of the *Optimized Adaptive Cruise Control* case study. The leading vehicle alternates between accelerating and braking as shown by the acceleration (lower left), and aims at keeping the velocity between 8 m/s and 12 m/s (upper right). The position (upper left) of our host car shows that it tightly follows the leading car, but always keeps a minimal distance that is considered safe. The failure rate (lower right) shows the aforementioned issue, where the initial state violates the monitor condition due to imprecision, but afterwards does not violate the monitor condition anymore.

In contrast to the optimized version, the unoptimized version is simpler and provides nondeterministic assignments for braking and acceleration. Moreover, the implementation switches accelerations only between the maximum braking force  $B$  and maximum acceleration force  $A$ . Although safety invariants still held at execution, we considered this behavior to be less convenient for human drivers. We realized that optimizing this behavior is best addressed in the modeling phase, as we decided to add additional modes for more fine-grained control.

In this specific case, we considered that resolving nondeterministic choice at implementation level to realize the same optimization would be significantly more difficult for two reasons. First, modes and their implementation in SL provide a *local* view on the parameters involved (e.g., velocity), whereas the implementation is much more detailed and scattered. The unoptimized version sets the acceleration to a fixed constant. The optimized version, however, views the acceleration as a function depending on variables and parameters, such as velocity, distance to leading vehicle, and worst-case execution time. Considering optimality of such functions increases effort during the implementation phase. Second, verifying correctness of such optimization is also more promising in the modeling phase, as numerical optimization is best addressed with differential dynamic logic.

The second example illustrating the *safe halt* case study is shown in Figure 4.6. The vehicle must keep a maximum velocity of 10 m/s and starts 40 meters in front of an obstacle. Similar to the unoptimized adaptive cruise control case study, the vehicle only switches accelerations between the maximum braking force  $B$  and maximum acceleration force  $A$  (upper left). The failure rate (lower right) shows an increase in monitor violations after four seconds. Although we



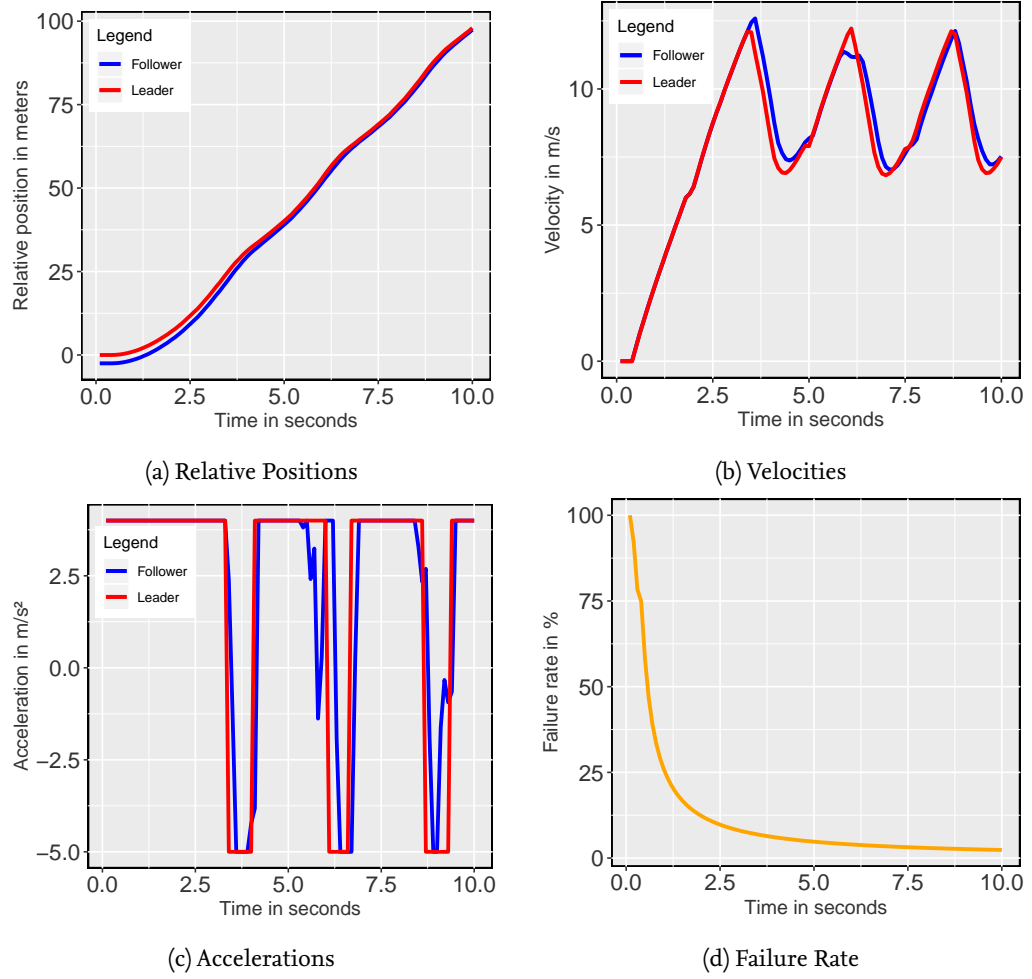


Figure 4.5.: Measurements for the *Adaptive Cruise Control* case study with the leading vehicle keeping a velocity between 8 and 12 m/s.

did not exactly locate the cause for this issue, as both safety guarantees were not violated, we assume that sensor uncertainties may play a role.

While failure rates should not be dismissed, the above experiments increase our confidence in the proposed link from verified skill-graph models to actual implementations. All evaluated maneuvers transferred their safety guarantees to the simulation. Although the first two case studies required more implementation effort (e.g., due to interfacing with AirSim and processing sensor data), the automatic code generation and reuse of existing implementations allowed us to implement the final three case studies considerably faster. We therefore believe that our verification and validation pipeline is particularly valuable for virtual prototyping of maneuvers and experimenting with them.

#### 4.4.3. Threats to Validity

Results of our evaluation are confronted with several threats to validity that we discuss in the following.

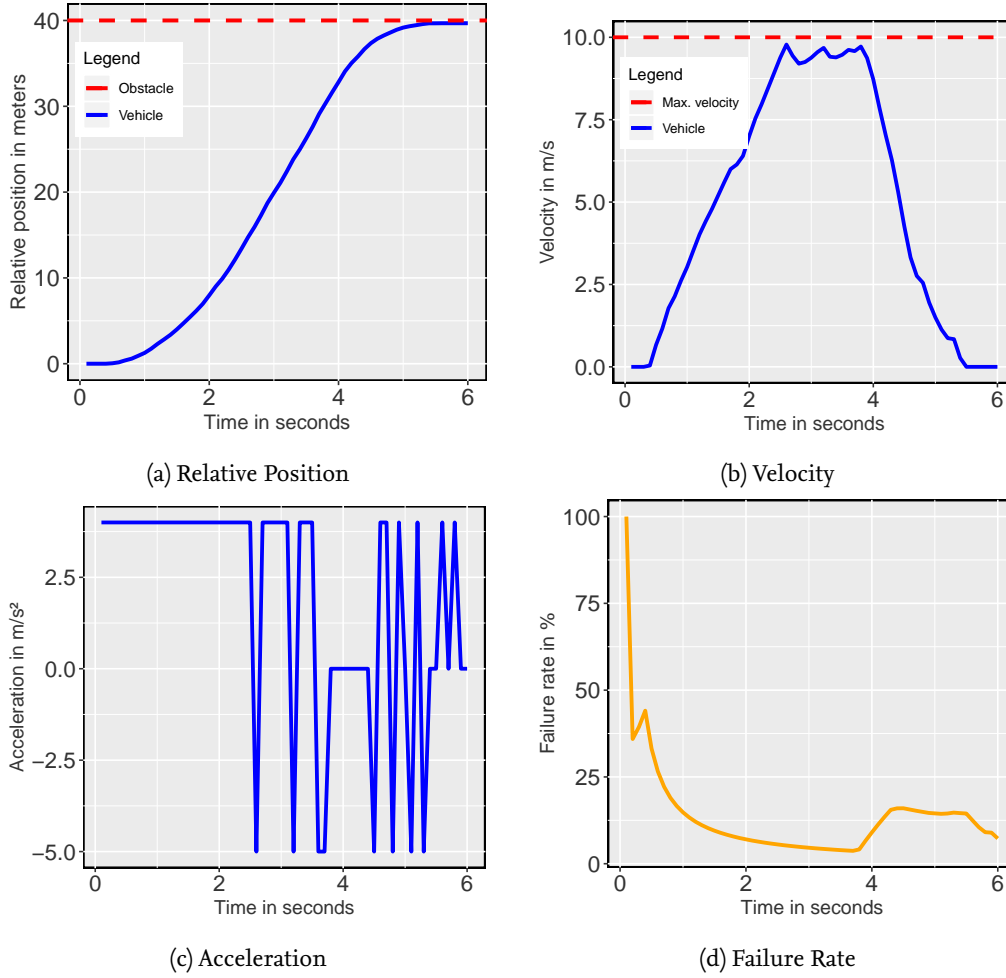


Figure 4.6.: Measurements for the *Safe Halt* case study with a maximum velocity of 10 m/s.

**Internal Validity.** The maneuvers that we evaluated may not represent real maneuvers as applied in practice. First, all of our maneuvers have relatively small model sizes and code bases. We acknowledge the fact that we only focus on stripped down versions of maneuvers for assessing controller correctness without fallback routines, as the goal of this thesis was not the development of production-ready maneuver implementations. Second, the majority of the case studies focus only on either the lateral (i.e., *lane keeping assistance*) or longitudinal aspect (i.e., *safe halt* and *adaptive cruise control*) of driving behavior. We tried to overcome this limitation by also providing a version of the *explore world* maneuver adapted for vehicles with larger turning circles. That is, *explore world* combines several aspects of driving behavior, while avoiding collision at all costs.

Reported failure rates greatly depend on the specifications we used, which may be incomplete with respect to real-world driving maneuvers. In particular, we mainly focused on single safety guarantees, such as collision freedom, maximum velocity, or deviation of a lane's center. Again, the *Explore World* and *Safe Halt* case studies combined two safety guarantees each, namely respecting a maximum velocity and avoiding collision. Moreover, we showed that safety guarantees of verified models hold at the level of implementation for each of the five case studies, and expect the same results for more complex maneuvers, such as the follow mode from the previous chapter.

Although we used ARCHICoRC for promoting the correct-by-construction approach, it could not be successfully applied as-is in our pipeline due to its lack of support for floating-point reasoning. We are therefore confronted with the threat of faulty implementations. We addressed this threat by manual inspection of the code. Moreover, safety goals were not violated during execution, which increases our confidence that no major defects were part of the derived implementations. As our case studies were rather simple, most nondeterministic assignments could be replaced by a single concrete assignment used for simulation. However, most of our maneuvers are therefore highly unoptimized. Additionally, we applied simple sensing provided by AirSim for avoiding collisions and manually provided the ground truth of the lane for the *Lane Keeping Assistance* case study. This way, manual implementation was reasonable.

**External Validity.** It is questionable to which extent our results can be generalized to more complex driving maneuvers, as we only evaluated five basic maneuvers. While more complex maneuvers would potentially lead to new insights, deductive verification of such maneuvers is still challenging and takes considerable effort. For instance, the *Adaptive Cruise Control* needed user interaction during verification, as the underlying model is optimized. Furthermore, implementing software skills performing tasks with respect to computer vision and artificial intelligence introduce additional bias, as these tasks are active research areas. Regardless of this threat, our results illustrate that our verification and validation pipeline works in principle for non-trivial maneuvers. Furthermore, most related work in this domain use the same level of complexity for their evaluations [A. Müller et al. 2018b; Loos et al. 2011; Bohrer et al. 2018].

## 4.5. Related Work

We already presented an overview of model-based verification and validation approaches for cyber-physical systems in [Chapter 1](#). In the following, we focus our discussion on related work that is close to the content of this chapter, namely the correctness-by-construction approach, component-based design with support for contracts, and end-to-end verification of cyber-physical systems.

### Refinement Calculi and the CoRC Ecosystem

The goal of refinement calculi is to provide a logical foundation for the stepwise construction of programs starting with an abstract specification. Pioneers in this field, who originally put the idea of stepwise program construction and correctness of program transformation forward, included Dijkstra [1972], Wirth [2001], Gerhart [1975], and C. A. R. Hoare [1969]. In this work, we applied the correctness-by-construction approach as proposed by Dijkstra [1976] and Kourie et al. [2012] to implement methods of provided interfaces. Related to the correctness-by-construction approach is Event-B [Abrial 2010], a framework which provides a formal automata-based language together with a notion of refinement to derive machine-executable code. The most prominent tool support to develop and reason about programs in Event-B is given by the RODIN platform [Abrial et al. 2010]. Although the correctness-by-construction approach to programming and Event-B pursue the same goal, they differ in their level of abstraction. Event-B abstracts from concrete programming, which requires additional care with respect to soundness in the code generation process,

whereas the correctness-by-construction approach results directly in compilable source code. Other refinement-inspired tools specifically targeting programs include ARCANGEL [Oliveira et al. 2003], which is based on Morgan’s refinement calculus [Morgan 1994] (i.e., an extension to the original refinement calculus with many more refinement rules) and SOCOS [Back et al. 2007; Back 2009], which mainly considers invariants instead of pairs of precondition and postcondition.

To enable the correctness-by-construction approach in our tool chain, we integrated CORC [Runge et al. 2019a] into ARCHICORC. The reason for this decision are manifold. First, the feature set and future plans of CORC are sufficient for the purpose of ARCHICORC. In particular, both tools target object-oriented languages and the small kernel of CORC’s theoretical foundation increases trust in its correctness. Second, CORC and ARCHICORC are based on the same technology stack, namely the *Eclipse Modeling Framework*. This leads to easier maintenance of the bridge between both tool suits and provides better user experience, as ARCHICORC artifacts and CORC programs can all be part of the same module. Third, CORC is well-maintained and actively developed, whereas most other frameworks in the field of stepwise program construction are not maintained anymore and also never reached a level of maturity, which we would consider sufficient enough for proper integration into ARCHICORC.

In spite of its young age, CORC was already extended in several directions. First, Runge et al. [2020] extended CORC with a notion of information flow control-by-construction. Instead of checking confidentiality of data post-hoc by static information-flow analyses, information flow control-by-construction defines refinement rules for constructing secure programs. Second, Bordis et al. [2020] introduced VARCORC, which is an offspring of CORC that focuses on correctness-by-construction for software product lines, instead of only considering monolithic programs. Finally, ARCHICORC, as presented in this chapter, lifts CORC to an architectural level by bundling correct-by-construction implementations in software components and providing means for code generation.

## Architectural Design with Support for Contracts

ARCHICORC aims at scaling the correctness-by-construction paradigm to large software projects by organizing CORC programs in a UML-style component-based architecture and providing means for source-code generation. In particular, it follows a UML-style modularization to separate concerns and to focus on reusable components. Consequently, the ARCHICORC’s component model is simpler compared to most of the component models applied in practice. Many contract theories are applied to embedded systems and component-based system’s engineering with a focal point on heterogeneity [Henzinger et al. 2007; Rawat et al. 2015] (i.e., contract theories that unify multiple functional and non-functional domains, such as software, hardware, mechanical, and electrical parts). Typically, component models in such theories [Benveniste et al. 2007; Benveniste et al. 2009; Sangiovanni-Vincentelli et al. 2012] comprise a set of input and output data ports (instead of interfaces) and a component behavior that relates such input and output streams. Contracts mainly follow the assume-guarantee paradigm [Benveniste et al. 2018]. In contrast to ARCHICORC, these components are typical dynamic in the sense that they follow a trace-based execution semantics.

There exist other tools and frameworks besides ARCHICORC that aim at combining UML-style component modeling with architectural specifications following the design-by-contract principle. Some examples worth mentioning include CBABEL [Rademaker et al. 2005], RADL [Reuss-

ner et al. 2003], XCD [Ozkaya et al. 2014] and its extension for visual modeling VXCD [Ozkaya 2017], and X-MAN [Lau et al. 2012]. In comparison with most of these tools, the purpose of our modeling approach differs. Their primary focus is to abstract away from any concrete programming model, whereas our focus is to organize correct-by-construction programs, which are constructed using a general-purpose programming language, and eventually generate a correct-by-construction architecture.

Furthermore, many contract-based frameworks propose to specify temporal properties as part of their assumptions and guarantees [Cimatti et al. 2015; Cimatti et al. 2012], including means for checking *contract refinement*. A popular implementation is given by the OCRA tool [Cimatti et al. 2013],<sup>10</sup> which leverages linear-time temporal logic to even specify properties sufficient for hybrid systems. OCRA has been integrated into larger tool chains and frameworks focusing on requirements formalization and contract-based architectural development, such as FoREVER [Baracchi et al. 2014] and AUTOFOCUS [Broy et al. 1999]. In contrast to ARCHICORC, these frameworks aim at industrial-size architectural models and employ model checking to verify their properties. It is certainly interesting to consider, which role the development process proposed in this and the previous chapter could take in such complex frameworks to focus on safety properties only.

## Simulation

Simulation is an integral part of our verification and validation pipeline. There exist numerous simulation environments to choose from, which all come with advantages and drawbacks. Most popular in the research community, GAZEBO [Koenig et al. 2004] is a simulation platform that offers a modular design that allows developers to integrate different physics engines and to create complex robotic systems with arbitrary sensor models and simple 3D worlds. Furthermore, GAZEBO maintains a close relationship with the robot operating system (ROS) [Quigley et al. 2009; Koubâa et al. 2017], one of the most prominent open-source frameworks for personal and industrial robotic systems. Therefore, GAZEBO is typically used for simulating systems based on ROS modules. Although GAZEBO comes with numerous features to increase realism in simulations, its rendering engine cannot compete with engines such as the Unreal engine or Unity, which makes it difficult to create visually-rich environments close to real-world scenarios.

To focus on visually-rich environments, AIRSIM [Shah et al. 2018] is a recent platform based on the Unreal engine that focuses primarily on automotive vehicles and flying drones. Most appealing, AirSim comes with pre-existing physical models of automotive vehicles and numerous 3D worlds (e.g., urban neighborhood or city), which saves development time and reduces the risk of introducing insufficient physical behavior. Especially, using independently-developed models and simulation environments is necessary in our evaluation to not invalidate empirical results.

In our validation and verification pipeline, we currently integrate both simulation platforms mentioned before. We primarily use GAZEBO/ROS for robotic systems, such as the vacuum cleaner robot introduced in the previous chapter (see Section 3.1.2), and AirSim for automotive vehicles, such as applied in our evaluation for various driving maneuvers. Moreover, the modularity of our combined tool suite (i.e., SKEDITOR and ARCHICORC) allows to extend the current set of simulation environments and to add new ones in a plug-and-play fashion.

<sup>10</sup><https://ocra.fbk.eu/>

## 4.6. Chapter Summary

The goal of this chapter was to develop a proof-of-concept methodology for moving verified skill graphs from the conceptual to the executable. Our focal point was twofold. First, one main concern was to retain correctness guarantees throughout the refinement process. Second, the other main concern was to also focus on important software engineering aspects, such as reusability and modularity. We argue that focusing on these concerns is necessary for enabling adaptation of our methodology in real software engineering processes.

In particular, we sketched a verification and validation pipeline for cyber-physical systems, where we explained how abstract maneuvers represented by skill graphs can be refined to a component-based architecture amenable to simulation and validation. With the correctness-by-construction approach, including its component-based manifestation ARCHICORC, we propose a framework that guides developers in deriving implementations that hold correctness guarantees for parts that cannot be generated automatically. Introducing an architectural intermediate layer supported by the design-by-contract paradigm can drastically increase reuse and reduce verification effort and implementation defects. Nevertheless, in its current stage, our verification and validation pipeline is only proof-of-concept and considerably depends on the features provided by the underlying third-party tools. We simulated the derived controller implementations in AirSim to inspect the appropriateness of the abstract model of a maneuver. The pursued and accomplished goal is that the link from a formal skill-graph model to execution is achievable in practice for non-trivial maneuvers. We have evaluated that all five case studies indeed transferred their correctness guarantees to the execution stage.

Although the correctness-by-construction approach can render post-hoc verification obsolete, it also comes with the disadvantage of increased specification and development effort, which is why our methodology allows both approaches. However, there exist two key problems for us in the context of deductive program verification, when it comes to adoption in software engineering processes. First, how can we assess precision of our our specification? Second, how can we maximize automation and efficiency of deductive program verifiers? We aim to address both of these questions in the next two chapters.

# 5. A Study on Mutation Analysis for Software Contracts

*This chapter shares material with the FormaliSE'21 paper “How much Specification is Enough? Mutation Analysis for Software Contracts” [Knüppel et al. 2021a].*

As indicated in the previous chapter, automatically verifying the correctness of source code requires a sufficient specification of the intended behavior. In post-hoc verification, which is also supported by ARCHICORC (see [Chapter 4](#)), we prove specification compliance only after the implementation was completed. This poses a major challenge for software developers working with deductive verification, as specifying software *precisely* is considered to be difficult, prone to error, and time consuming [Baumann et al. 2012; Hähnle et al. 2019; Gleirscher et al. 2018]. Due to low adoption rates, lack of expertise and tool support complicate matters further [Gleirscher et al. 2018; Gleirscher et al. 2019].

As a consequence, developers often tend to not specify all but only the simplest *properties* that their implementation must satisfy. In this case, multiple *diverging implementations* may be covered by the same specification and therefore labeled as *correct*.<sup>1</sup> This misleading sense of correctness is the root cause for numerous software quality problems. First, method invocations can be abstracted with their respective contract [Ahrendt et al. 2016; Knüppel et al. 2018b]. If the contract, however, does not cover properties which the caller relies on, automatic verification becomes typically impossible. As a solution, either (1) contracts have to be adapted, (2) a different, sufficiently specified method has to be invoked instead, or (3) the implementation has to be inlined instead of relying on contract abstraction. The drawbacks are wasted development time, implementation clones, and decreased performances during verification. Second, during software evolution, developers that create or modify methods must always be aware of imperfect specifications of callees, which leads to code smells [Van Emden et al. 2002]. Third, software bugs can be overlooked if parts of the implementation are unspecified. Again, a misleading sense of correctness may even reinforce the emergence of critical software bugs, as testing effort is typically reduced.

While an exact method to compute the distance between specification and implementation is desirable, to the best of our knowledge no such method does currently exist. Moreover, this problem has gained little attention in the research community. Consequently, when proposing novel analyses, there currently exists no adequate baseline to compare against. In this chapter, we investigate this topic with a study on *how to quantify the precision of software contracts* by means of *mutation analy-*

---

<sup>1</sup>We use the term *diverging* to mean that multiple implementations have same method signature and return type but exhibit different observable behaviors.



sis [Papadakis et al. 2019; Jia et al. 2010; Offutt et al. 2001]. Originally, mutation analysis (or *mutation testing*) in the context of software development is used to evaluate the quality of existing test suites by performing small modifications on programs called *mutations* [DeMillo et al. 1978]. The goal is to check to what extent the test suite is oblivious to these modifications. If too many mutants *survive* (i.e., are not *killed*), the test suite is typically of low quality and needs improvement. In justification of this evaluation approach, DeMillo et al. [1978] postulated two hypotheses:

- *Competent Programmer*: This hypothesis states that the majority of software bugs introduced by senior developers can be attributed to small syntactic mistakes rather than conceptual fallacies.
- *Coupling Effect*: This hypothesis states that there often exist a coupling of small and complex mistakes, such that identification of one small mistake may lead to the discovery of other, more severe problems.

We assume that both hypotheses hold similarly for contract-based software, as there exists a correspondence between test cases and software contracts. That is, valid test cases specify instances of correct functional behavior, which is also the goal of software contracts. We argue that the key question of this chapter (i.e., *whether a mutation analysis is a promising technique to compute a contract's strength*) is worth to discuss. In particular, our mutation analysis applies mutation operators that we adopted from the literature [Ma et al. 2006; Hou et al. 2007] to programs and method contracts. Afterwards, the *mutation score* is computed, which is defined as the percentage of killed mutants and therefore is able to approximate a contract's strength. Furthermore, mutants that survive constitute examples of unspecified behaviors and can be further inspected by developers.

Although we are confident that our insights apply to other programming and specification languages, we consider again JAVA programs with JML contracts [Leavens et al. 2006] and employ the deductive program verifier KEY 2.6.3 [Ahrendt et al. 2016] to prove contract compliance automatically. In Section 5.1, we first illustrate the challenges raised above on a motivating example. In Section 5.2, we give a definition of contract strength and present a taxonomy of causes for incomplete contracts. In Section 5.3, we propose a mutation analysis framework for measuring a contract's strength and then empirically evaluate this framework in Section 5.4.2. Finally, we discuss related work on measuring the precision of specifications in Section 5.5.

## 5.1. Motivating Example

We exemplify the problem and consequences of incomplete specifications raised above by means of a contract in JML from the open-source project *Paycard* [Engel et al. 2010]. The *Paycard* application allows users to create paycards with user-defined limits and provides operations for charging them. Furthermore, all charging attempts are logged in a singleton structure called *LogFile*, where each such entry is represented by class *LogRecord*.

In Listing 5.1, we show an excerpt of class *LogRecord*, which declares method *setRecord* for initializing its members. The precondition following the keyword **requires** assumes that argument *balance* and class member *transactionCounter* are non-negative integers. After executing method *setRecord*, the postconditions following the keyword **ensures** then guarantee that

---

```

1 package paycard;
2 public class LogRecord {
3     /*@ public invariant !empty ==> (balance >= 0 && transactionId >= 0);
4     @*/
5
6     private /*@spec_public@*/ static int transactionCounter = 0;
7     private /*@spec_public@*/ int balance = -1;
8     private /*@spec_public@*/ int transactionId = -1;
9     private /*@spec_public@*/ boolean empty = true;
10
11     /*@ public normal_behavior
12         @ requires balance >= 0 && transactionCounter >= 0;
13         @ ensures this.balance == balance &&
14             @ transactionId == \old(transactionCounter);
15         @ ensures transactionCounter >= 0;
16         @ assignable empty, this.balance, transactionId, transactionCounter;
17     @*/
18     public void setRecord(int balance) throws CardException{
19         if(balance < 0){
20             throw new CardException();
21         }
22         this.empty = false;
23         this.balance = balance;
24         this.transactionId = transactionCounter++;
25     }
26     //[...]
27 }

```

---

Listing 5.1: Excerpt of class LogRecord from the *Paycard* case study to illustrate incomplete specifications (adopted from [Knüppel et al. 2021a]).

(1) field `this.balance` equals argument `balance`, (2) field `transactionId` is set to the value that class member `transactionCounter` held prior to method execution, and (3) the value of class member `transactionCounter` remains non-negative. The frame condition represented by the keyword **assignable** states that only the four fields `empty`, `this.balance`, `transactionId`, and `transactionCounter` are allowed to be written to by method `setRecord`. Finally, the *class invariant* following the keyword **invariant** must hold before and after each method execution and enforces that both fields `balance` and `transactionId` remain non-negative after object creation as long as boolean field `empty` evaluates to false.

Employing the deductive program verifier KEY [Ahrendt et al. 2016], contract compliance of method `setRecord` is automatically provable, which may increase trust in the method’s correctness. At the same time, there exist other diverging implementations that also satisfy the illustrated specification. That is, method `setRecord` is *underspecified*, which consequently enables the masking of unintended defects by allowing non-equal implementations to adhere to the same

specification. First, the second postcondition guarantees that class member `transactionCounter` remains non-negative after method execution (Line 15). The implementation, however, increments `transactionCounter` by one each time method `setRecord` is called on an object of type `LogRecord` (Line 24). Therefore, eliminating the increment operator in the implementation constitutes a valid implementation, but certainly stays in conflict with the intended behavior (i.e., that `this.transactionId` must be unique). Second, method `setRecord` sets field `empty` to false (Line 22), which is not specified. As a consequence, the class invariant (Line 3), which depends on the valuation of field `empty`, becomes useless in terms of modular reasoning; any method that depends on `setRecord`'s contract is oblivious to the fact that field `empty` is set to false.

As emphasized in this example, underspecification of method contracts comes with at least two major drawbacks. First, although correctness of method `setRecord` is automatically provable, *imprecise* (or incomplete) contracts can mask unintended implementation errors and therefore provide a misleading sense of safety with respect to the verification result. For instance, uniqueness of field `transactionId` is not guaranteed. Second, modular reasoning on the basis of contracts is impaired. In the context of deductive verification with the purpose of saving verification effort, it is typically desirable to replace method calls with corresponding contracts instead of inlining implementations [Ahrendt et al. 2016; Knüppel et al. 2018b]. However, this requires that contracts are precise enough. For instance, `setRecord`'s contract does not specify that field `empty` is set to false. Proving correctness of another method that invokes method `setRecord` and relies on this fact will only be possible if the invoked method is inlined. `setRecord`'s contract is therefore less applicable as a callee in the context of other methods.

Both drawbacks pose major obstacles for developers who want to follow the design-by-contract principle and apply deductive verification in their software projects. The aim of this chapter is to investigate to what extent a mutation analysis can help to identify *incomplete contracts* and also to quantify their incompleteness. In the next section, we define explicitly what we mean by *incomplete contracts* and discuss their causes.

## 5.2. A Taxonomy for Incomplete Specifications

Before presenting our mutation analysis approach in the next section, we must first give a definition of a contract's strength and explore various causes for incomplete specifications. Although we are confident that our considerations transfer to other programming and specification languages as well, we again focus on JAVA and JML. In Section 5.2.2, we give a definition of the *completeness* and *strength* of a method contract as considered in this chapter. In Section 5.2.2, we discuss several causes for incomplete contracts and set the focus for the remainder of this chapter.

### 5.2.1. A Definition of Contract Strength

With *contract strength*, we refer to how precisely a method contract covers the corresponding implementation. That is, the more diverging implementations comply to the exact same specification, the *weaker* the contract is. We give an illustration of this intuition by means of a Venn-diagram in Figure 5.1.

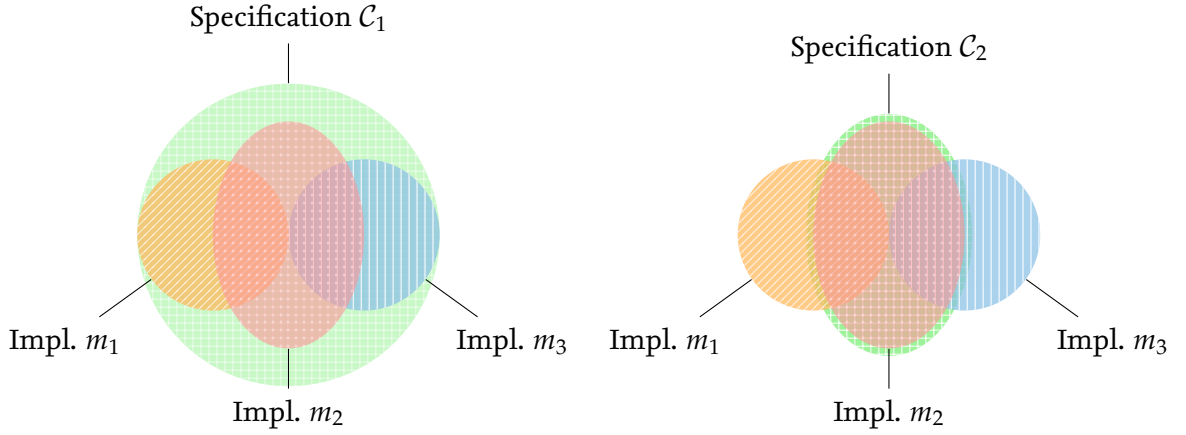


Figure 5.1.: Venn-diagram illustrating the difference in *completeness* of two specifications for diverging implementations. The shapes display allowed behaviors of either the specification or implementation (adopted from [Knüppel et al. 2021a]).

The figure abstractly presents two method contracts abbreviated with  $C_1$  and  $C_2$ , and three method implementations abbreviated with  $m_1$ ,  $m_2$ , and  $m_3$ . As can be seen on the left, all three divergent implementations  $m_1$ ,  $m_2$ , and  $m_3$  are covered by the shape of specification  $C_1$ , which means that their correctness with respect to  $C_1$  is provable. In contrast, the shape of specification  $C_2$  is specifically tailored to implementation  $m_2$ , which means that only implementation  $m_2$  complies to  $C_2$ . Comparing both contracts, we can say that contract  $C_1$  is *weaker* than contract  $C_2$  (i.e., it satisfies more divergent implementations), and, conversely,  $C_2$  is *stronger* than contract  $C_1$  (i.e., it satisfies less divergent implementations). If the goal implementation is  $m_2$ , then specification  $C_2$  is typically desired; it reflects the expected behavior more precisely and prevents software quality issues as described in the beginning of this chapter.

Following the notation of the previous chapter, we again denote by  $c = \{\phi\}m\{\psi\}$  a *method contract*, where  $\phi$  and  $\psi$  are the precondition and postcondition, respectively, and  $m$  is an abbreviation for a method signature. For the sake of presentation, we denote by  $\mathcal{C}$  the universe of all method contracts (i.e.,  $c \in \mathcal{C}$ ) and by  $\mathcal{M}$  the universe of all method signatures (i.e.,  $m \in \mathcal{M}$ ). Furthermore, we introduce set  $\mathcal{I}_m$  that represents a countable set of *all* implementations with observably different behavior for a given method signature  $m \in \mathcal{M}$  in some programming model. That is, any two implementations  $I_1, I_2 \in \mathcal{I}_m$  differ syntactically and in behavior, but are both valid for method signature  $m$ . Satisfaction between implementation and contract follows again the typical notion for dynamic logic (see Section 2.3). That is, for method contract  $c$ , we write  $\models \phi \rightarrow [I]\psi$  to express that executing implementation  $I$  satisfies postcondition  $\phi$  in the post-state when precondition  $\psi$  holds prior to execution. We then give a notion for completeness of contracts with the following definition.

#### Definition 5.1: Completeness of Contracts

Let  $c = \{\phi\}m\{\psi\}$  be a *method contract*. We say that contract  $c$  is **complete** if all implementations that satisfy  $c$  are behaviorally equivalent (i.e.,  $|\{I \in \mathcal{I}_m \mid \models \phi \rightarrow [I]\psi\}| = 1$ ). Otherwise, we say that contract  $c$  is **incomplete** (i.e.,  $|\{I \in \mathcal{I}_m \mid \models \phi \rightarrow [I]\psi\}| \geq 2$ ).

We assume that most non-trivial contracts in practice are incomplete, as otherwise specification effort would be too high and cost-ineffective. Measuring the exact degree of completeness, however, is generally either infeasible due to scalability issues or even undecidable to the underlying problem of program equivalence. Intuitively, the incompleteness of a method contract  $\{\phi\}m\{\psi\}$  corresponds to the cardinality of the set of valid method implementations (i.e.,  $\{I \in \mathcal{I}_m \mid \models \phi \rightarrow [I]\psi\}$ ). Unfortunately, this set can become too large to compute if it is computable at all. An easier subproblem to address is to identify which of two method contracts is more complete (i.e., *stronger*) than the other.

#### Definition 5.2: Relative Contract Strength

Let  $c = \{\phi\}m\{\psi\}$  and  $c' = \{\phi'\}m\{\psi'\}$  be two method contracts for the same method signature  $m$ . If  $c$  is satisfied by a strict subset of observably different behaviors compared to  $c'$

$$\{I \in \mathcal{I}_m \mid \models \phi \rightarrow [I]\psi\} \subset \{I \in \mathcal{I}_m \mid \models \phi' \rightarrow [I]\psi'\}, \quad (5.1)$$

then we say that contract  $c$  is **stronger** (i.e., more complete) than contract  $c'$ . Conversely, we say that contract  $c'$  is **weaker** (i.e., less complete) than contract  $c$ .

Relative strength between contracts is easier to compute, as it can be reduced to the problem of behavioral subtyping [Liskov et al. 1994]. That is, a contract  $\{\phi\}m\{\psi\}$  is stronger than a contract  $\{\phi'\}m\{\psi'\}$ , iff both conditions  $\phi \Rightarrow \phi'$  and  $\psi' \Rightarrow \psi$  hold. As mentioned above, we will apply a mutation analysis to heuristically compute a representative sample of set  $\{I \in \mathcal{I}_m \mid \models \phi \rightarrow [I]\psi\}$ , where both definitions are needed. Before that, we present a classification scheme for incomplete contracts in the next subsection.

### 5.2.2. A Classification of Incomplete Contracts

The motivating example in the previous section highlighted that incomplete contracts are often-times caused by underspecified postconditions. However, there exist other causes for incompleteness, such as unnecessarily strong preconditions. Moreover, modern specification languages, such as JML [Leavens et al. 2006] or ACSL [Baudin et al. 2008], are *richer* in syntax and allow to express properties besides precondition and postcondition in their contracts. Consequently, one may think of additional categories that confine the causes for an incomplete contract more precisely. In Figure 5.2, we present a coarse-grained taxonomy of causes for incomplete contracts. The taxonomy comprises four key causes, namely (A) *Unnecessarily strong precondition*, (B) *too weak postcondition*, (C) *underspecified exceptional behavior*, and (D) *framing condition*, which we discuss in the following.

**(A) Unnecessarily Strong Precondition.** Although reports show that unnecessarily strong preconditions are rare in practice [A. F. Milanez et al. 2013; A. Milanez et al. 2017], they constitute causes for incomplete specifications by being too restrictive. Such preconditions lead to weaker contracts than necessary, as they require less input to be supported. From a caller's perspective, Meyer [1992] argues that strong preconditions for invoked methods are preferred, as they explicitly limit the number of scenarios in which a method is safe to be called and, therefore, better support developers. At the same time, it can be argued that too strong preconditions unnecessarily limit applica-

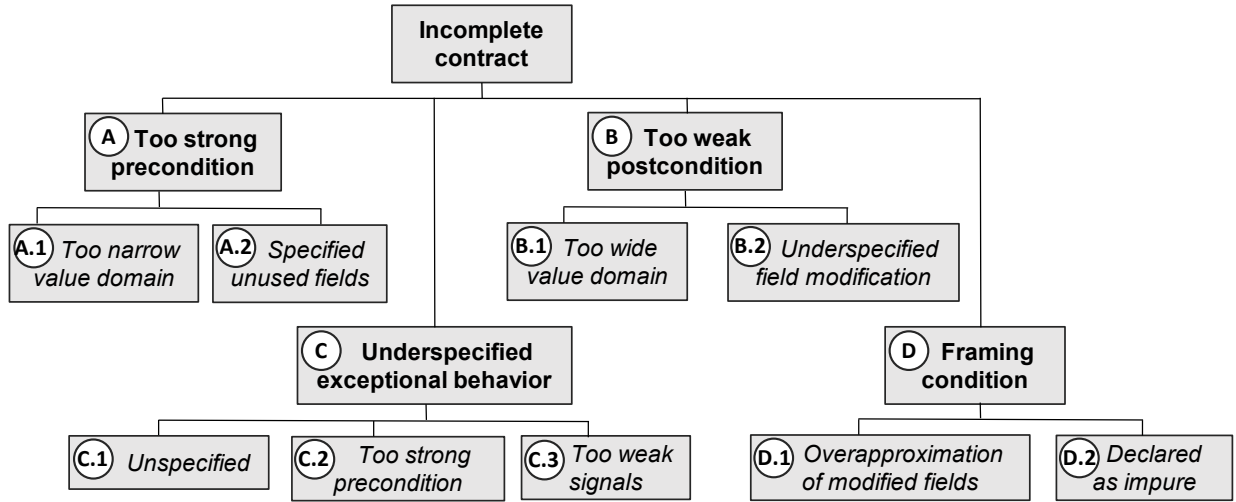


Figure 5.2.: Taxonomy of causes for incomplete software contracts for object-oriented programming languages (adopted from [Knüppel et al. 2021a]).

bility of *contract abstractions*, which then may require method inlining instead for successful automated verification. Furthermore, such preconditions may confuse developers due to their unjustified restrictiveness, which impairs software quality in general. In particular, too strong preconditions stem either from too narrow value domains of accessed arguments and fields (A.1), or from fields that are restricted to specific values, but not accessed at all (A.2). In both cases, adequate analyses can be valuable to developers to identify and be aware of these instances.

**(B) Too Weak Postcondition.** Preconditions make assumptions about the state prior to method execution. In contrast, postconditions give guarantees about the state after method execution. That is, postconditions have a stronger relation with the method’s *implementation* (i.e., observable behavior), as they are *asserted* during verification, whereas preconditions merely specify value ranges of used fields and arguments. Indeed, in the context of design by contract and deductive verification, contract  $\{\phi\}m\{\psi\}$  is equivalent to contract  $\{\text{true}\}m\{\phi \Rightarrow \psi\}$ , where the precondition is incorporated into the postcondition. Therefore, we hypothesize that underspecification of postconditions is the main cause for incomplete contracts. Similar to stronger-than-necessary preconditions, if callers rely on the specification of a contract, the contract must specify all details of its behavior necessary for the caller. Otherwise, applicability is again reduced. Furthermore, as motivated in the beginning of this chapter, underspecified behavior can easily mask critical bugs and provide a false sense of correctness. Analogous to unnecessarily strong preconditions, too weak postconditions stem from either too wide value domains for modified fields (B.1), or from particular field modifications that are not specified or at least underspecified (B.2). Within the latter category, we also include that specific *properties* are not specified sufficiently. For instance, consider a sorting method that gets as input an integer array and returns as output an integer array with elements sorted in ascending order. Specifications of such methods should typically specify at least two properties: (1)



the output array must be sorted in ascending order and (2) the output array must be a permutation of the input array. If the latter property is missing, field modifications are underspecified.

**(C) Underspecified Exceptional Behavior.** Additionally to specifying observable behavior of methods under normal execution, contracts may also specify instances, in which exceptions are raised. These cases are specified similar to normal specification cases; a precondition states when the respective exception is raised, the postcondition that must hold in the post-state is specified following the keyword **signals**, and the keyword **signals\_only** is used to specify the type of exception that is thrown. If exceptions are raised but their instances are not adequately specified, it can be argued that the overall contract is incomplete. In particular, causes are similar to the previous two paragraphs: either the raised exception is not specified at all (C.1), the precondition is stronger than necessary (C.2), or the postcondition following the keyword **signals** is weaker than possible (C.3).

**(D) Imprecise Framing Conditions.** In Listing 5.1, we used the keyword **assignable** to establish a framing condition [Hatcliff et al. 2012; Ahrendt et al. 2016], which is a set-theoretic notion for listing all fields that are allowed to be modified by the respective method. Framing conditions constitute helpful annotations for verifiers, as they provide information to which degree a state may change and are therefore often indispensable for automatic verification when method calls are abstracted with their contracts. In the context of deductive verification with KEY, omitting the **assignable** clause is identical to **assignable \everything**, which is added implicitly and means that any field modification is allowed. In contrast, **assignable \nothing** is used to explicitly forbid any field modifications. As a special case, JML also allows to mark methods as *pure* [Leavens et al. 2006; Helm et al. 2018] with the **pure** modifier, which means that such methods are side-effect free and either terminate or raise an exception.<sup>2</sup> In JML and KEY, pure methods are allowed to be queried in contracts, which increases their applicability in contract-based software projects overall. Based on these considerations, two causes with respect to framing result in incomplete contracts. First, the **assignable** clause may list fields that indeed remain unchanged during any method execution (D.1). This may again unnecessarily complicate automatic verification of callers. Second, methods that may be considered as pure are implicitly marked as *impure* (D.2). As explained above, this reduces their applicability, either in other pure methods or in contracts themselves.

In the framework and conducted study that we present in the following, we primarily focus on the first two categories, namely *unnecessarily strong preconditions* (A) and *too weak postconditions* (B). We argue that both categories constitute the prime targets for a mutation analysis. First, results from these two categories can probably be applied to exceptional specification cases, which are similar in design. Second, for framing conditions, we suspect that data-flow analyses have higher precision and potentially scale better [Helm et al. 2018; Yu et al. 2020]. In the next section, we present a mutation-based framework to analyze completeness of method contracts based on the considerations of this section.

<sup>2</sup>In KEY, the **pure** modifier is shorthand for the combination **assignable \nothing** and **diverges false**, where the **diverges** clause is used for specifying either partial (**true**) or total correctness (**false**) [Ahrendt et al. 2016].



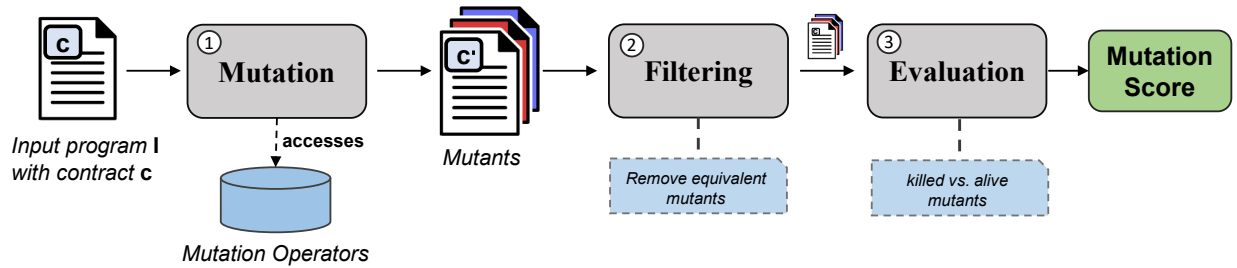


Figure 5.3.: Schematic workflow of our mutation analysis for contract-based software (adopted from [Knüppel et al. 2021a]).

## 5.3. Mutation Analysis for Software Contracts

In the previous section, we mentioned that automatically calculating the exact subset of observably different behaviors all satisfying the same method contract is, in general, practically infeasible. We therefore follow the methodology of *mutation testing* [Papadakis et al. 2019; Jia et al. 2010; Offutt et al. 2001] with the goal to calculate a representative sample of this subset. In particular, we aim at mutating a given method implementation or method contract to generate observably different behaviors with the hope that this allows developers to evaluate the precision of their method contracts. First, we give an overview of our mutation analysis framework for contract-based software in Section 5.3.1. Next, we present the chosen mutation operators in Section 5.3.2 and discuss three interesting software metrics for developers resulting from our mutation analysis in Section 5.3.3. Finally, we discuss limitations of our approach in Section 5.3.4.

### 5.3.1. Overview of the Mutation Analysis

We illustrate the schematic workflow of a mutation analysis in Figure 5.3, which takes as input a specified method and a set of predefined *mutation operators*, and outputs the *mutation score* as a measurement for the strength of the method’s contract. Besides input and output, the workflow consists of three consecutive phases, namely the *mutation phase* ①, the *filtering phase* ②, and the *evaluation phase* ③. We briefly discuss each of the three phases in the following.

In the mutation phase ①, the *mutator* accesses a set of pre- and well-defined *mutation operators* and applies each of them (or a reasonable subset) to the source method. The goal of mutation operators is to systematically make syntactic changes to the source program in order to create *mutants* that still constitute compilable programs. For instance, consider an operator that aims at replacing arithmetic operations systematically. An expression  $a + b$  in the source program could then be changed to any expression given by the set  $\{a - b, a/b, a\%b, \dots\}$  in one of the mutants. Moreover, one mutation operator generally leads to numerous mutants. First, each identified instance in the source program (e.g., each arithmetic expression) becomes subject to modification and leads to a new and syntactically unique mutant. Second, the granularity of an operator (i.e., the kind of changes the operator makes) is up to the developers. That is, an operator for replacing arithmetic operations may choose between numerous replacements instead of creating a unique operator for

each particular replacement. We discuss the mutation operators used in our framework and their granularity in [Section 5.3.2](#). Formally, we give the following definition for a mutation operator.

**Definition 5.3: Mutation Operator**

Let  $c = \{\phi\}m\{\psi\}$  be a *method contract* and  $I \in \mathcal{I}_m$  a method implementation for  $m$ . A **mutation operator** is a function

$$op_m : \mathcal{C} \times \mathcal{I}_m \rightarrow 2^{\mathcal{C} \times \mathcal{I}_m}, (c, I) \mapsto \{(c_i, I_i)\}_i$$

that takes a pair of a method contract and method implementation  $(c, I)$  as input and produces a (possibly empty) set of new pairs  $\{(c_1, I_1), \dots, (c_n, I_n)\}$  called **mutants** as output. We denote by  $\mathcal{O}_m$  the set of mutation operators for method signature  $m$ .

A particularity of our definition is that mutation operators are not limited to the implementation, but may also mutate method contracts. We postpone the discussion on the usefulness of mutating method contracts to [Section 5.3.2](#) and mostly consider source-code mutation in this subsection. Then, applying each mutation operator in  $\mathcal{O}_m$  to the source program results in a set of mutants, for which we give the following formal definition.

**Definition 5.4: Set of Mutants**

Let  $c = \{\phi\}m\{\psi\}$  be a *method contract* and  $I \in \mathcal{I}_m$  a method implementation for  $m$ . The **set of mutants** is defined as

$$\mathbb{M}_{(c,I)} = \{op_m(c, I) \mid op_m \in \mathcal{O}_m\} \setminus \{(c, I)\}.$$

A prominent problem with mutation analyses is that of *equivalent mutants* [[Grün et al. 2009](#)], which must be removed during the filtering phase ②. Equivalent mutants are mutants that may differ in syntax, but behave semantically equal. For instance, consider a return statement **return**  $i$ ; which is semantically equivalent to the return statement **return**  $i++$ ;. Although a mutation operator may add arithmetic operations to variables, the increment operator in the latter code snippet is without effect.

In particular, there are two kinds of equivalent mutants that can distort the final mutation score of an analysis. First, mutants that are equivalent to the source program will naturally survive and count towards the mutation score, but not lead to new insights. Second, generated mutants that behave semantically equal will also skew the mutation score, either to the better or the worse depending on their survival. However, due to the connection to the problem of program equivalence, identifying equivalent mutants is one of the hardest problems in mutation analysis to address [[Grün et al. 2009](#)]. In this thesis, we consider their identification as an orthogonal problem and therefore rather point to the rich corpus of research that deals with equivalent mutants [[Grün et al. 2009](#); [Madeyski et al. 2013](#); [Papadakis et al. 2015](#); [Kintis et al. 2017](#)]. In our empirical study, we identified equivalent mutants by a manual inspection of all mutants. For convenience, filtering equivalent mutants is already part of [Definition 5.4](#). First, mutation operators are defined over the set of observably different implementations  $\mathcal{I}_m$ , which means that equivalent mutants map to the same

implementation. Second, if a mutant is equivalent to the pair consisting of the source contract and source implementation, it is removed from the set of mutants.

In the evaluation phase ③, we try to verify each mutant automatically, where each mutant is either successfully verified (representing an *alive* mutant) or *killed* in the process. In practice, the outcome of the evaluation phase per mutant is indeed threefold. First, verification of the mutant is successful. Then, the respective mutant stays alive and constitutes another valid implementation that likewise complies to the method contract. Second, the mutant cannot be verified. That is, the verifier is unable to find a proof either through timeout or because the algorithmic search does not know how to progress. In this case, the mutant is killed and counted towards the number of eliminated mutants. Third, the mutant is not compilable. The reason is that some mutation operators may produce syntactically incorrect code, which we may report on, but otherwise do not consider any further. Again, we omitted a notion for syntactically incorrect implementations in [Definition 5.3](#) and [Definition 5.4](#).

Finally, a *mutation score* based on the number of killed and alive mutants is calculated, which ideally is a sufficient approximation of a contract's strength. Calculating the mutation score follows the standard literature [[Offutt et al. 2001](#)]. That is, the mutation score is defined as the quotient between the number of killed mutants and the number of non-equivalent mutants. How close the mutation score is to the true strength of a contract  $\{\phi\}m\{\psi\}$  mostly depends on the mutation operators themselves and their capability to approximate the set of observable different behaviors  $\mathcal{I}_m$ . Formally, we define the mutation score as follows.

#### Definition 5.5: Mutation Score

Let  $c = \{\phi\}m\{\psi\}$  be a *method contract* and  $I \in \mathcal{I}_m$  the method implementation for  $m$ . The **mutation score** is defined as

$$\text{MutationScore}(c, I) = \frac{|\{(c, I') \in \mathbb{M}_{(c, I)} \mid \phi \not\models [I']\psi\}|}{|\mathbb{M}_{(c, I)}|}.$$

Next, we discuss the mutation operators that we apply in our mutation analysis.

### 5.3.2. Mutation Operators

The mutation operators we selected for our mutation analysis are standard operators obtained from the literature, which we divide into *source-code operators* [[Ma et al. 2006](#)] for imperative languages (e.g., JAVA or C++) and *contract-level operators* [[Hou et al. 2007](#)] for contract-based specification languages with explicit preconditions and postconditions (e.g., JML or ACSL). In [Table 5.1](#), we give a list of source-code and contract-level mutation operators that are available in our framework. We discuss both kinds of mutation operators in the following.

#### Source-Code Mutation Operators

The top 16 mutation operators presented in [Table 5.1](#) are applied directly to the implementation and partitioned into seven different categories:

Category	Operator	Description
<b>Arithmetic</b>	AOR	Replaces arithmetic operators
	AOI	Adds a unary operator to any variable
	AOD	Removes a unary operator
<b>Relational</b>	ROR	Replaces relational operators
<b>Conditional</b>	COR	Replaces binary condition operators
	COI	Negates conditional expressions
	COD	Removes negation from conditional expressions
<b>Shifting</b>	SOR	Replaces shift operators
<b>Logical</b>	LOR	Replaces bitwise operators
	LOI	Adds bitwise complement to any variable
	LOD	Removes bitwise complement from a variable
<b>Assignment</b>	ASRS	Replaces assignment operators
<b>Deletion</b>	SDL	Deletes a statement
	VDL	Deletes all occurrences of a referenced variable
	CDL	Deletes a constant
	ODL	Deletes an arbitrary operator from an expression
<b>Contract</b>	PW	Weakens precondition
	PS	Strengthens postcondition

Table 5.1.: List of available source-code and contract-level mutation operators (adopted from [Knüppel et al. 2021a]).

**Arithmetic:** The arithmetic category focuses on arithmetic operators, such as  $\{+, ++, -, --, *, /, \%\}$ , and consists of a mutation operator for exchanging arithmetic operators (AOR), adding a unary operator, such as arithmetic negation (i.e.,  $-$ ) or the decrement operator (i.e.,  $--$ ), to a variable (AOI), and removing unary operators (AOD). Example for AOR:

$$a + b \mapsto \{a - b, a/b, a * b, \dots\}$$

**Relational:** The relational category focuses on relational operators, such as  $\{>, >=, <, <=, ==, !=\}$ , and only consists of a mutation operator for exchanging relational operators (ROR). Example for ROR:

$$a > b \mapsto \{a < b, a <= b, a >= b, a == b, \dots\}$$

**Conditional:** The conditional category focuses on conditional operators, such as  $\{!, \&\&, ||\}$ , and, similar to the arithmetic category, consists of mutation operators for exchanging conditional operators (COR), adding the unary negation to formulas (COI), and removing it from formulas (COD). Additionally, COR may also choose to replace a binary subexpression with either `true` or `false`. Example for COR:

$$a \&\& b \mapsto \{a || b, \text{true}, \text{false}, \dots\}$$

**Shifting:** The shifting category is dedicated to shifting operators, such as  $\{\ll, \gg\}$ , and only consists of a mutation operator for exchanging shifting operators (SOR). Example for SOR:

$$a \ll b \mapsto \{a \gg b\}$$

**Logical:** The logical category focuses on bitwise operators, such as  $\{\&, |, ^, \sim\}$ , and, again, consists of mutation operators for exchanging bitwise operators (LOR), adding the bitwise complement (i.e.,  $\sim$ ) to expressions (LOI), and removing it from expressions (LOD). Example for LOR:

$$a \& b \mapsto \{a | b, a \wedge b, \dots\}$$

**Assignment:** The assignment category is dedicated to short-cut assignment operators, such as  $\{+=, -=, \dots\}$ , and consists only of a mutation operator for exchanging them (ASRS). Example for ASRS:

$$a += b \mapsto \{a -= b, a *= b, \dots\}$$

**Deletion:** The deletion category focuses on either deleting complete statements or on removing parts of statements. The category consists of mutation operators for deleting complete statements (SDL), deleting variables and references (VDL), deleting constants from expressions (CDL), and deleting operators (ODL). Example for SDL:

$$a += b \mapsto \emptyset$$

### Contract-Level Mutation Operators

Besides means for mutating the implementation, we also provide mutation operators for mutating contracts. In [Table 5.1](#), we list two contract-level mutation operators, namely PW for *weakening* preconditions and PS for *strengthening* postconditions. We argue that both operators can be valuable for developers to identify better contracts. First, as our starting position is a verified method implementation, strengthening the postcondition through slight syntactic changes may identify corner cases that eventually help to improve the specification. Moreover, a surviving mutant already represents a direct example of a stronger contract that can be inspected manually. Second, weakening preconditions allows developers to inspect whether the original preconditions can be improved to increase applicability and software quality.

Both mutation operators PW and PS are inspired by Hou et al. [2007], who investigated mutation of interface specifications for black-box testing. As mutation of specifications is rather rare in practice, both operators come with an interesting research question that is worth to investigate: *given the mutation rules for specifications proposed by Hou et al. [2007], do they complement source-code mutation or provide valuable insights for contract-based projects?* We address this question in our empirical study in [Section 5.4](#). Before, we describe the mutation rules for both operators in more detail.

To apply contract-level mutation operators in our projects, we assume that contracts are *normalized*. Normalized contracts have only one precondition and one postcondition, which must both be in *conjunctive normal form*. That is, a precondition  $\phi$  is normalized iff it has the form  $\phi \equiv \bigwedge_{i=0}^n \phi_i$ ,

Precondition Weakening (PW)		Postcondition Strengthening (PS)	
Original	Mutant	Original	Mutant
$==$	$\geq, \leq$	$\geq$	$>, ==$
$>$	$\geq, !=$	$\leq$	$<, ==$
$<$	$\leq, !=$	$!=$	$>, <$
$P \geq Q$	$P \geq Q - 1$	$P > Q$	$P > Q + 1$
$P \leq Q$	$P \leq Q + 1$	$P < Q$	$P < Q - 1$
$\&\&$	$  $	$  $	$\&\&$
forall	exists	exists	forall

Table 5.2.: List of contract mutation rules as proposed by Hou et al. [2007].

where each  $\phi_i$  is a subformula of the form  $\phi_i \equiv \bigvee_{j=0}^m l_{i,j}$ .  $l_{i,j}$  represent atomic expressions over variables, fields, and quantifiers (i.e., forall and exists).

In Table 5.2, we list the exact mutation rules for both contract-level operators as proposed by Hou et al. [2007]. All illustrated rules are applied to all atomic expressions  $l_{i,j}$  for either weakening the precondition (i.e., PW) or strengthening the postcondition (i.e., PS). The provided mutation rules only apply small and inexpensive syntactic changes to contracts that are known to result in stronger contracts. However, these syntactic changes must be applied carefully, as both operators may also lead to the counterpart of equivalent mutants for contract-level mutations. That is, some resulting mutants may exhibit a weakened instead of a strengthened contract compared to the original one, which is trivially satisfied. An optimization to rule out equivalent mutants is that only non-negated atomic expressions of postconditions are mutated. We illustrate this optimization with the following example.

**Example 5.1.** Consider the postcondition  $\psi \equiv a \geq b \Rightarrow \text{result} == a$ , where the value of variable  $a$  is returned if it is greater than or equal to the value of variable  $b$ . Normalizing  $\psi$  leads to  $\psi_{\text{norm}} \equiv !(a \geq b) \vee \text{result} == a$ . Applying mutation operator PS on  $\psi_{\text{norm}}$  results in the mutated postcondition  $\psi'_{\text{norm}} \equiv !(a > b) \vee \text{result} == a$ . It follows that  $\neg(a \geq b) \equiv a < 5 \Rightarrow a \leq b \equiv \neg(a > b)$ . That is, applying PS on negated atomic expressions weakens postconditions, which leads to mutants that are trivially satisfied. Typically, such expressions result from either implications in formulas or from range conditions used in quantifier expressions, and must be considered during mutation. In particular,  $\neg(a \geq b)$  is equivalent to  $a < b$ , for which no mutation rule exists for operator PS in Table 5.2.

### 5.3.3. Three Software Metrics for Contract Incompleteness

As mentioned in the beginning of this chapter, there exists a strong correspondence between software testing and formal verification of contract-based software. In both disciplines, mutation analysis has the goal to improve either test cases or contracts. We can argue that mutating contracts seems to be the dual to mutating test cases, which implies the question: *what insights do we gain by mutating test cases?*

Contract-level mutation aims at strengthening specifications, as weaker specifications per definition cover less properties to verify and are consequently trivially satisfied. In particular, mutating the precondition means to require that more inputs are supported by the corresponding im-

plementation. Alive mutants reveal that more inputs are valid for the same implementation, which is an important analysis result for developers. Analogously, mutating the postcondition decreases the number of accepting states after method execution. An alive mutant reveals that the implementation delivers more behavior than is covered by the corresponding contract. Again, it is important for developers to recognize this mismatch between contract and implementation. In particular, there is a clear separation between source-code mutation and contract-level mutation. The former focuses rather on closing the gap between implementation and specification, whereas the latter directly focuses on slightly improving the specification.

Based on these considerations, we argue that it is worth to provide three distinct metrics as outcome of a mutation analysis as opposed to a combined mutation score. The three metrics result from (1) mutating the source code, which is a metric for the *contract's strength* (see mutation score in [Definition 5.5](#)), (2) only mutating the precondition, which corresponds to the *looseness* of the assumed state prior to execution, and (3) only mutating the postcondition, which measures the degree of *underspecification* of the postcondition. The three metrics are computed as follows (derived from computing the general mutation score; see [Definition 5.5](#)) for a pair of method contract  $c$  and a method implementation  $I$ .

■ **Metric 1 – Contract's strength (Source-code mutation):**

$$\text{Score}_{CS}(c, I) = \frac{|\{(c', I') \in \mathbb{M}_{(c, I)} \mid \phi \not\models [I']\psi' \wedge c \equiv c'\}|}{|\{(c', I') \in \mathbb{M}_{(c, I)} \mid c \equiv c'\}|}$$

■ **Metric 2 – Looseness of precondition (Precondition mutation):**

$$\text{Score}_{LP}(c, I) = \frac{|\{(c', I) \in \text{PW}(c, I) \mid \phi' \not\models [I]\psi'\}|}{|\text{PW}(c, I)|}$$

■ **Metric 3 – Underspecification of postcondition (Postcondition mutation):**

$$\text{Score}_{UP}(c, I) = \frac{|\{(c', I) \in \text{PS}(c, I) \mid \phi' \not\models [I]\psi'\}|}{|\text{PS}(c, I)|}$$

### 5.3.4. Soundness and Completeness

We briefly discuss limitations of our mutation analysis for contract-based software that we presented in this section so far.

It is important to note that weak specifications are primarily caused by weaker-than-necessary postconditions. In contrast, preconditions are not enforced and simply assumed before verification takes place. That is, the program verification checks whether the postcondition holds after execution when the precondition held before. If the precondition is violated, contract compliance is trivially satisfied. However, such methods are still callable at run-time with any input arguments, therefore leading to unpredictable behavior. Alive mutants resulting from operator PW can therefore not be directly considered as *improved* or *stronger* contracts. We argue that these instances must be inspected by developers, who must assess manually whether unnecessarily strong preconditions are formulated with intention or should be refactored.



Regarding completeness of our operators, we acknowledge that the scope of both contract-level mutation operators PW and PS is limited to only small syntactic changes. The drawback is that only small improvements can be identified, whereas we assume that the majority of weak postconditions stems from the absence of complete properties. Our hope is that our study provides a baseline for further improvements in this line of research, which is why we only concentrated on mutation operators proposed by other researcher. In the future, it is possible to develop more powerful operators based on contract inference [Ernst et al. 2007; Polikarpova et al. 2009; Wei et al. 2011] and compare their effectiveness with our results.

## 5.4. Evaluation

So far, we emphasized the correspondence between test cases and software contracts, and proposed to use a mutation analysis to heuristically evaluate completeness of specifications. In [Section 5.3.3](#), we formulated three software metrics (i.e., *contract strength*, *looseness of precondition*, *under-specification of postcondition*) resulting from a mutation analysis, which may provide valuable insights to developers working with contract-based software.

A number of questions arise that we address in this section. Do weak specifications occur in practice? How well can our proposed mutation analysis identify weak specifications? Are there mutation operators that are more effective than others? Are the contract-level mutation operators useful at all? We aim to answer these questions by an empirical study. First, we describe our prototypical open-source implementation in [Section 5.4.1](#). Next, we introduce the research questions and methodology of this evaluation in [Section 5.4.2](#). In [Section 6.6.2](#), we present and discuss our results. Finally, in [Section 6.6.3](#), we discuss potential threats undermining the validity of our study.

### 5.4.1. Prototypical Implementation

Within this chapter, we study whether a mutation analysis provides insights to improve specifications. To the best of our knowledge, sufficient tool support for applying a mutation analysis as described in [Section 5.3](#) for JML-based software projects does not exist, which is why we developed a prototype for our experimental evaluation. In particular, such tool support must (1) incorporate a mutation framework for JAVA/JML programs and (2) employ an *evaluator*, such as a deductive program verifier or model checker for JML programs, to verify conformance between specification and implementation.

For mutating JAVA/JML programs, we chose the open-source mutation framework  $\mu$ JAVA [Ma et al. 2006], which already provides all source-code mutation operators described in [Table 5.1](#). Additionally, we extended  $\mu$ JAVA with the two contract-level mutation operators PW and PS. For evaluating JAVA/JML mutants, we employ the deductive program verifier KEY-2.6.3 [Ahrendt et al. 2016], which is able to verify many specified method implementations automatically. [Figure 5.4](#), gives an overview of the architecture.

First, the implementation allows end-users to configure which mutation operators are applied to which input programs via dedicated *configuration files* ①. We illustrate such a configuration file in [Listing 5.2](#). Additionally, we provide more fine-grained versions of source-code mutation operators in our implementation. For instance, operator AOR is further refined to operators AORB, deal-

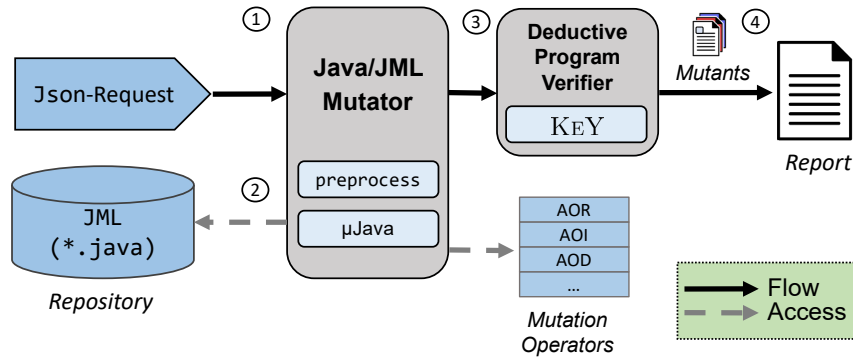


Figure 5.4.: Architecture of the prototypical implementation of our mutation analysis framework (adopted from [Knüppel et al. 2021a]).

```

1 {
2   "MuJava_Home": "path/to/muJava",
3   "Source": "study/BankAccount/Account.java",
4   "Method": "int_getAccountNumber()",
5   "MethodSignature": "public int getAccountNumber()",
6   "ProjectPath": "study/BankAccount",
7   "Operators": [
8     "AORB", "AORS", "AOIU", "AOIS", "AODU", "AODS", "ROR",
9     "COR", "COD", "COI", "SOR", "LOR", "LOI", "LOD", "ASRS",
10    "SDL", "VDL", "CDL", "ODL", "PW", "PS"
11  ]
12 }

```

Listing 5.2: Json-request for mutating a method from the BankAccount case study.

ing with binary arithmetic operations (e.g.,  $a+b$ ), and AORS, dealing with unary arithmetic operations (e.g.,  $a--$ ). In line with the workflow described in Section 5.3.1, the JAVA/JML mutator starts with normalizing the requested method contracts in a preprocessing step ②. Afterwards,  $\mu$ JAVA is employed to generate the mutants based on the initial configuration. As explained in Section 5.3.3, we do not combine source-code and contract-level mutations in our mutation score, but treat them differently. Next, we employ KEY-2.6.3 to check validity of all generated mutants ③ and generate a final report ④ with respect to the three metrics proposed in Section 5.3.3.

Our prototypical tool support is available online<sup>3</sup> and enables users to add new mutation operators, either to mutate the source code or to mutate the specification. While identifying equivalent mutants is crucial for the results we obtain in our study, the current development stage of our prototype provides no means for identifying equivalent mutants automatically. We therefore resort to manual inspection and leave automated support for identifying equivalent mutants for future work.

## 5.4.2. Methodology and Evaluated Projects

Based on our prototypical tool support for performing a mutation analysis on JAVA/JML projects, it is now possible to discuss the questions raised above. For this, we collected a total of ten open-source JML projects that we use as experimental subjects in our evaluation. In Table 5.3,

<sup>3</sup>Available at <https://github.com/TUBS-ISF/MutationAnalysisForDBC-Formalise21>.

Project	Classes	Methods	Where?
Dnivra	1	3	JMLOK2 [A. Milanez et al. 2014]
ShorreWaitAlgorithm	1	7	
Paycard	2	6	
BankAccount	7	19	
BankAccountV2	2	6	SPL2GO <sup>4</sup>
PaycardSPL	2	5	
ExampleFromChapter7	2	7	KEY [Ahrendt et al. 2016]
ExampleFromChapter16	2	3	
OpenJML	8	20	OpenJMLDemo <sup>5</sup>
DutchFlagAlgorithm	1	1	Kourie et al. [2012]
<b>Total</b>	<b>28</b>	<b>77</b>	

Table 5.3.: Evaluated open-source JML projects (adopted from [Knüppel et al. 2021a]).

we summarize all experimental subjects used in our evaluation, including the number of classes, number of specified and evaluated methods, and origin. In particular, we aim at discussing the following four research questions.

**RQ-1:** *To what extent do incomplete specifications occur in our projects?*

**RQ-2:** *To what extent is a mutation analysis able to measure completeness of contracts in JML programs?*

**RQ-3:** *Which source-code mutation operators are more effective than others?*

**RQ-4:** *What insights do the contract-level mutation operators provide in our subjects?*

We hypothesize that the majority of incomplete specifications arises from either missing field modifications or more complex unspecified properties, which we are unable to identify with the mutation rules of our contract-level mutation operators (see Section 5.3.2). At the same time, completeness of contracts is one of the most important metrics, as it addresses the gap between specification and implementation, whereas the contract-level mutation only aims at slightly improving specifications. Consequently, research questions **RQ-1** – **RQ-3** are discussed by only applying source-code level mutations (i.e., investigating metric  $\text{Score}_{CS}$  for a contract’s strength). In particular, the *average* mutation score per project  $P$ , where  $\mathbb{M}_P$  is the set of all mutants of project  $P$ , is calculated as

$$\text{Score}_{SC}^P = \frac{\sum_{(c,I) \in \mathbb{M}_P} \text{Score}_{SC}(c, I)}{|\mathbb{M}_P|}.$$

The impact of contract-level mutation is addressed separately in **RQ-4**.

To answer **RQ-1** and **RQ-2**, we apply our mutation analysis to each specified method in our experimental subjects. All investigated methods with a perfect mutation score (i.e.,  $\text{Score}_{CS} = 1$ ) are inspected manually to identify overlooked incomplete specifications (i.e., false negatives). This way,

<sup>4</sup><http://spl2go.cs.ovgu.de>

<sup>5</sup><https://github.com/OpenJML/OpenJMLDemo>

Project	Methods	Mutants	Alive Mutants	Score <sub>CS</sub> <sup>P</sup> ( $\emptyset$ )	Analysis / Ground Truth
Dnivra	3	25	0	1.00	0 / 0
ShorreWaitAlgorithm	7	48	0	1.00	0 / 0
Paycard	6	132	5	0.96	1 / 2
BankAccount	19	135	59	0.56	4 / 6
BankAccountV2	6	261	24	0.79	4 / 6
PaycardSPL	5	113	6	0.96	1 / 2
ExampleFromChapter7	7	105	6	0.96	2 / 2
ExampleFromChapter16	3	152	1	0.99	1 / 1
OpenJML	20	315	59	0.81	7 / 8
DutchFlagAlgorithm	1	181	2	0.99	1 / 1
<b>Total</b>	<b>77</b>	<b>1,467</b>	<b>162</b>	<b>-</b>	<b>21 / 28</b>

Table 5.4.: Results of our mutation analysis per project (adopted from [Knüppel et al. 2021a]).

we are able to establish the ground truth, which we use as comparison in **RQ-2**. For **RQ-3**, we evaluate which mutation operators are most effective and which mutation operators are least effective for our subjects. This enables us to decide whether certain ineffective but costly operators can be ignored. Finally, for **RQ-4**, we evaluate whether contract-level mutation as proposed by Hou et al. [2007] provides valuable insights for our experimental subjects.

### 5.4.3. Results and Insights

We applied our mutation analysis as described in Section 5.3 to each specified method per project to generate the maximum number of possible mutants. That is, we did not limit the number of generated mutants to some maximum value and also did not consider a time-out for generating mutants, as overall execution time was bearable for our study. Syntactical mutants were excluded from our calculation. Equivalent mutants were filtered relying on a manual inspection. Although this bears the risk of overlooked equivalent mutants, we are confident that such rare instances will not distort the overall conclusions we draw from our evaluation.

In Table 5.4, we illustrate each project, the number of evaluated methods, the number of generated mutants and the number of alive mutants, the average mutation score per project, and the number of weak contracts identified by our mutation analysis in contrast to the ground truth. The average mutation score per project may give a feel for the overall quality of contracts per project. In Appendix A, we present all mutation scores for each analyzed method. In the following, we discuss all four research questions raised above.

### RQ-1: Incomplete Specifications in Subjects

With this research question, we investigate to what extent incomplete specifications occur in our experimental subjects. Table 5.4 shows that 28 methods, out of a total of 77 methods, are identified as *incomplete*. Only two of the ten projects, namely Dnivra and ShorreWaitAlgorithm, did not contain any noticeable incomplete contracts. Besides project DutchFlagAlgorithm, which contains

---

```

1  [...]
2  /*@
3    requires bArray != null;
4    requires offset >= 0;
5    requires offset < 32766; //prevents overflowing
6    requires 0 <= offset && offset < bArray.length - 1;
7    requires bArray.length >= 2;
8    ensures bArray[offset] == hour;
9    ensures bArray[offset+1] == minute;
10   assignable bArray[offset], bArray[offset+1];
11  */
12  public short getTime(byte [] bArray, short offset) {
13    short aux = offset;
14    bArray[aux++] = hour;
15    bArray[aux++] = minute;
16    return (short) (offset + (short) 2);
17  }
18  [...]

```

---

Listing 5.3: Underspecified method from class OpenJML.Time.

only a single method, BankAccountV2 contains the biggest portion of incomplete method contracts (100%), followed by OpenJML and PaycardSPL (both 40%).

Most incomplete specifications we identified stem from underspecification of the postcondition. To give an example, we illustrate the method `Time.getTime` from the OpenJML project in [Listing 5.3](#). Although the method returns a value in [Line 16](#), the result itself is not specified in the method's contract. Consequently, each mutant modifying [Line 16](#) in any way will still comply to the method contract and counted towards the number of alive mutants.

**Discussion.** The manual analysis of our experimental subjects illustrates that incomplete method contracts occur frequently in even smaller open-source JML projects. We assume that larger projects suffer even more from incomplete specifications. Method contracts in such projects are often not checked formally, but rather used to document the intended behavior of method explicitly. A reason for this can be limited tool support. We can therefore hypothesize that, once method contract and implementation conform to each other, developers are even more hesitant to improve contracts than to check them formally in the first place. This emphasizes that new techniques and automated tool support are needed to give developers better insights into their contract-based projects. In summary, we identified that roughly a third of all analyzed methods (36%) could be considered as incompletely specified.

## RQ-2: Effectiveness of Proposed Mutation Analysis

With this research question, we evaluate the effectiveness of our proposed mutation analysis. In [Table 5.4](#), we present (1) the number of incomplete specifications identified by the mutation anal-

---

```

1 /*@
2  @ normal_behavior
3  @ requires A.length > 0 && (\forallall int i; i>=0 &&
4      i<A.length; A[i] == 0 || A[i] == 1 || A[i] == 2);
5  @ ensures (\forallall int q; q >= 1 && q < \result.length;
6      \result[q-1]<=\result[q]);
7  @*/
8 public static int[] DutchFlag( int[] A ) {
9     int wb = 0, wt = 0, bb = A.length;
10    /*@ loop_invariant [...] @*/
11    while (wt != bb) {
12        if (A[wt] == 0) {
13            int t = A[wt];
14            A[wt] = A[wb];
15            A[wb] = t;
16            wt = wt + 1;
17            wb = wb + 1;
18        } else if (A[wt] == 1) {
19            wt = wt + 1;
20        } else if (A[wt] == 2) {
21            int t = A[wt];
22            A[wt] = A[bb % 1]; //original version: bb - 1
23            A[bb - 1] = t;
24            bb = bb - 1;
25        }
26    }
27    return A;
28 }

```

---

Listing 5.4: Alive mutant of the dutch national flag algorithm as presented by Kourie et al. [2012].

ysis and (2) the average mutation score per project. In line with our ground truth, all generated mutants for the ten methods of projects Dnivra and ShorreWaitAlgorithm were killed, leading to a mutation score of 100% for both projects. The majority of other projects have an average score of over 90%, namely Paycard, BankAccountV2, PaycardSPL, ExampleFromChapter7, and ExampleFromChapter16. The two projects with the highest number of methods, BankAccount and OpenJML, also have the lowest averaged mutation scores. In particular, a manual inspection of BankAccount revealed that most methods were only vaguely specified, resulting in a high number of alive mutants (59) and a relatively low mutation score (59%).

A particularly interesting instance is project DutchFlagAlgorithm, which is an algorithmic and specified solution to the *dutch national flag problem* [Dijkstra 1976] given by Kourie et al. [2012]. We show an alive mutant of its implementation in Listing 5.4. The challenge of the dutch national flag problem is to sort an integer input array, whose values are restricted to 0, 1, and 2 (numbers correspond to the colors red, white, and blue of the Dutch flag). While Kourie et al. [2012] men-

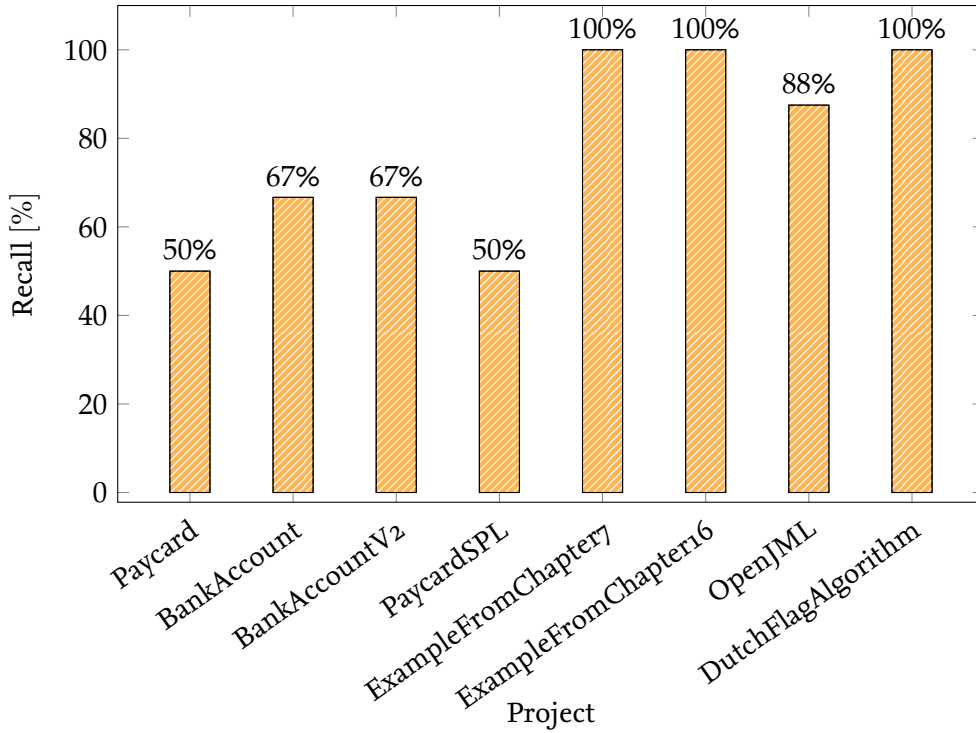


Figure 5.5.: Percentage of identified incomplete specifications compared to the ground truth (adopted from [Knüppel et al. 2021a]).

tioned that the permutation property is an integral aspect of specifications for sorting algorithms, the illustrated method contract in Listing 5.4 does only specify that elements in the output array are in ascending order. Two alive mutants caught this missing property, which is why the mutation score of `DutchFlagAlgorithm` is not 100%. Listing 5.4 shows one of these mutants, where mutation operator AORB replaced the binary subtraction with the modulo operator. For instance this mutant turns the input array `[1, 0, 2, 2, 1, 1, 2]` into output array `[0, 0, 1, 1, 1, 2, 2]`, which is a clear violation of the permutation property. However, numerous inputs to this mutant will still respect the permutation property, which emphasizes the evaluation of mutants with formal verification where every input is checked.

Overall, our mutation analysis was able to identify the majority of incomplete specifications (75%). In Figure 5.5, we present the percentage of identified incomplete specifications per project (i.e., *recall*). We identified that the lower values (e.g., project `Paycard` and `PaycardSPL`) were often due to a low number of true incomplete specifications combined with few generated mutants (i.e., an indicator for a method’s complexity). Projects with considerably larger code bases resulted in many more generated mutants (i.e., `BankAccount` and `OpenJML`) and also a higher recall (e.g., 88% for `OpenJML`).

**Discussion.** Most projects had a relatively high mutation score of over 90%. The reason is that these projects contained none (`Dnivra` and `ShorreWaitAlgorithm`) or only few true incomplete specifications (`Paycard`, `PaycardSPL`, `ExampleFromChapter7`, `ExampleFromChapter16`, and `DutchFlagAlgorithm`), and oftentimes identified weak specifications were strong enough to eliminate most generated mutants. However, our results show that a mutation analysis can provide valuable insights



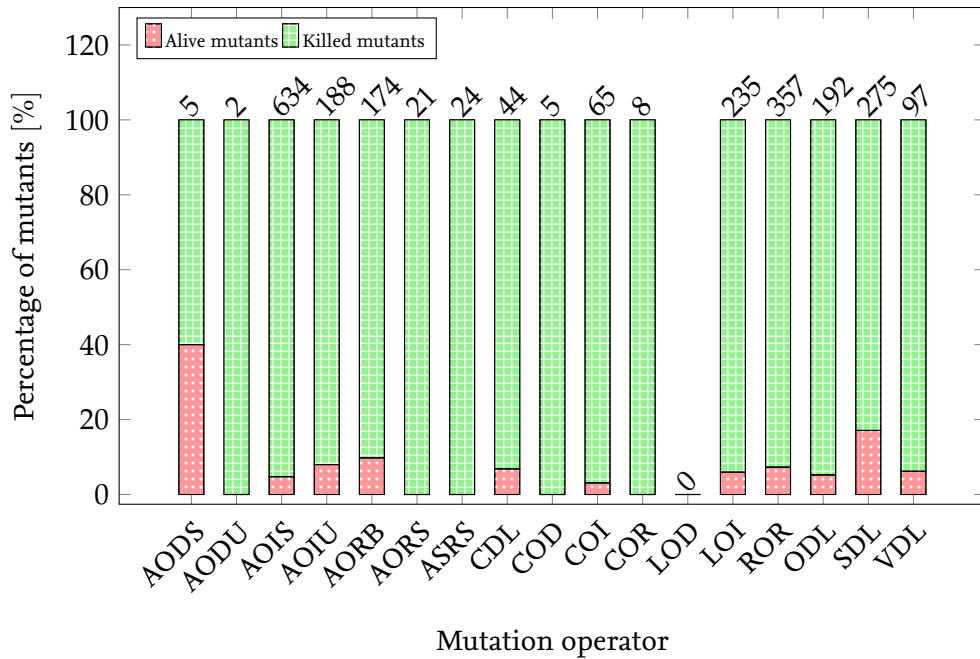


Figure 5.6.: Effectiveness of source-code mutation operators (adopted from [Knüppel et al. 2021a]).

for developers. First, analyzing project BankAccount resulted in the lowest averaged mutation score. Based on a manual assessment, project BankAccount can indeed be considered as the most vaguely specified project of our experimental subjects. Second, the largest and most complex subject is OpenJML, where our analysis managed to identify seven out of eight weak specifications (88%). Additionally, the averaged mutation score of 81% seems to be an adequate reflection of the average incompleteness of all contained method contracts. Finally, although the DurchFlagAlgorithm has a mutation score of 99%, our results show that even a small number of alive mutants can provide insights to incomplete specifications. This indicates that missing properties can be identified with a mutation analysis, but will likely result in high mutation scores. Means for automatically analyzing alive mutants seems to be a reasonable continuation of this line of work.

Although we acknowledge that our evaluation is limited in size and complexity, we hypothesize that mutation analyses for contract-based projects are most effective for larger and more complex projects. That is, the more mutants are generated, the higher the chance to identify divergent implementations that still comply to the specification.

### RQ-3: Effectiveness of Mutation Operators

With this research question, we investigate to what extent certain source code mutation operators are more effective than others. In Figure 5.6, we illustrate each source-code mutation operator and the total number of generated mutants, which we further divided into killed and alive mutants. In contrast to Table 5.4, we included equivalent mutants as well. The only mutation operator that produced zero mutants is LOD. The reason is that the bitwise complement was not part in any of our experimental subjects. Six other operators produced only few mutants, namely AODS, AODU, AORS, ASRS, COD, and COR. From the six operators, only AODS produced mutants that could be

verified. Most incomplete specifications were identified by operator SDL (i.e., 47 alive mutants), followed by operator AOIS (i.e., 30 alive mutants).

**Discussion.** Our results indicate that mutation operators leading to alive mutants can be effective for identifying weak specifications regardless of the number of mutants they generate. For example, operator AODS produced only five mutants, but was the only operator that identified the underspecification of `transactionCounter` from project Paycard illustrated in [Section 5.1](#). In contrast, mutation operator AOIS produced a total of 634 mutants, of whom 604 were killed (95%). Even worse, not a single mutant that survived led to a unique discovery, such with operator AODS above. That is, all 30 alive mutants were equivalent to another mutant produced by another operator. Based on our subjects, operator AOIS could be removed.

We conclude that an optimized set for our experimental subjects contains the following mutation operators: AODS, AOIU, AORB, ODL, ROR, and SDL. In particular, this set results in 1191 mutants (i.e., 49% less mutants generated), while identifying the exact same weak specifications as before. An interesting follow-up question is whether the optimized set leads to a more accurate mutation score. We are confident that the results obtained in this evaluation give rise for larger evaluations in the future to address such questions.

#### RQ-4: Contract-Level Mutation Operators

In this research question, we evaluate whether the contract-level mutation operators PW and PS lead to valuable insights. In [Figure 5.7](#), we depict the total number of generated and nonequivalent mutants per project and divided these mutants further into (1) *killed mutants*, (2) identified mutants with *unnecessarily strong preconditions* (i.e., produced by PW), and (3) identified mutants with *too strong postconditions* (i.e., produced by PS). In projects `ShorreWaitAlgorithm` and `PaycardSPL`, none of the generated mutants could be verified. Moreover, only a single mutant generated by operator PS in project Paycard could be verified. This is in line with our assumption that operator PS will lead to very few alive mutants, as the operator is constructed in a way to only catch slight deviations in specifications. In contrast, numerous unnecessarily strong preconditions were identified across the majority of projects with the help of operator PW. In particular, 31 mutants based on 16 method could be verified with a strengthened version of their preconditions.

**Discussion.** These results indicate that contract-level mutation operator PW supports developers in identifying preconditions that are not enforced on the implementation level. Although it can be argued that one goal of design by contract is to reduce *defensive programming* (i.e., handling of prohibited inputs at implementation level), violated preconditions do not prohibit that such methods are called at runtime with unexpected inputs. Consequently, identification of these occasions can be beneficial for developers. It can also be argued that operator PW complements the source-code mutation operators, which are only able to identify too weak postconditions. Operator PS did not have much impact in our evaluation, which is in line with our hypothesis that incomplete specifications are often due to missing properties. These are hard to identify with operator PS and easier to identify with source-code mutation. A future direction is, however, to develop more sophisticated contract-level mutation operators, whereas results and insights of our evaluation may serve as a baseline.

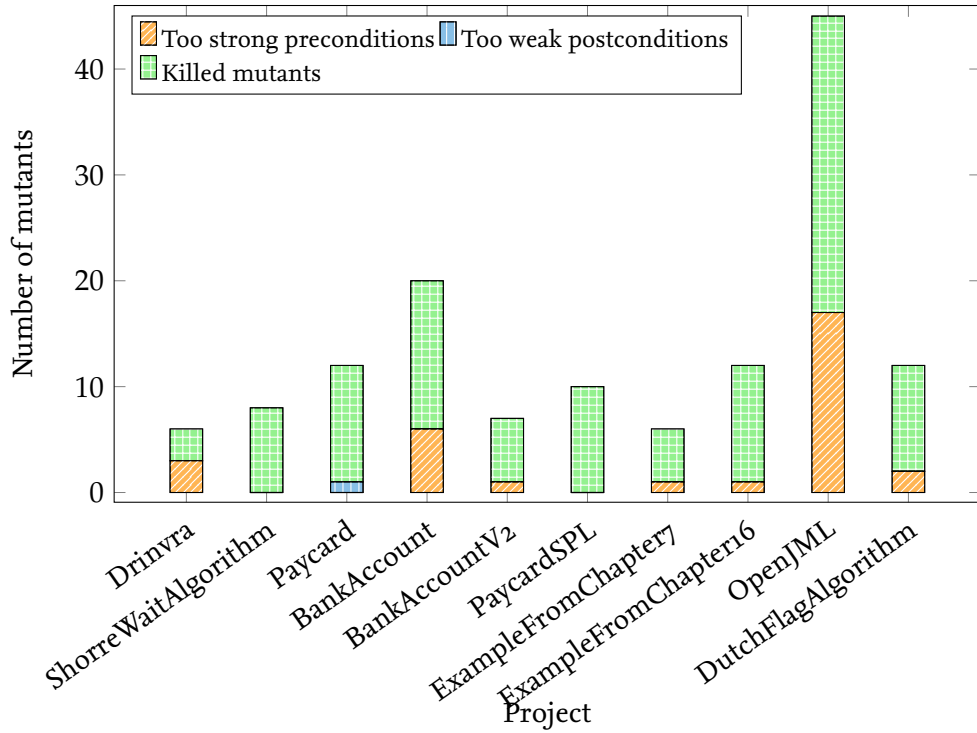


Figure 5.7: Number of contract-level mutants per project partitioned into killed and alive mutants resulting from operators PW and PS (adopted from [Knüppel et al. 2021a]).

#### 5.4.4. Threats to Validity

Results of our evaluation are confronted with several threats to validity that we discuss in the following.

**Internal Validity.** To establish the ground truth of incomplete specifications in our experimental subjects, we resorted to a manual inspection prone to human error. The limited number of projects (i.e., ten projects), methods (i.e., 77 method contracts) and their complexity allowed us to identify incomplete contracts in a reasonable amount of time. Furthermore, we manually strengthened and verified the identified instances, which increases our confidence in the truthfulness of our assessment. However, there is still the chance that we missed incomplete specifications. Again, the subjects we chose for this study were limited in size and complexity, which allowed us to perform a manual assessment in the first place. Furthermore, results of our mutation analysis came to similar conclusions. That is, every single incomplete contract identified by the mutation analysis was also identified by our manual assessment. For equivalent mutants, we additionally performed sanity checks in the form of test cases, where we randomly generated inputs and compared resulting outputs. Although there is still a chance that we missed cases, we argue that this would only marginally impact the conclusions we drew in this evaluation.

Another threat is *selection bias* with respect to the projects and method contracts we analyzed. In particular, the combined number of method contracts over all projects is much higher than the number of method contracts we analyzed (i.e., 77). The reason is that numerous methods could not be parsed successfully with KEY and were therefore excluded. We restrained from adapting

contracts that could not be parsed in order to not introduce any additional bias in our study. We acknowledge that large-scale evaluations are necessary to continue this line of research, but are also confident that this initial study constitutes a stepping stone towards that vision.

**External Validity.** Our evaluation is targeted towards a small number of JML projects, which puts generalizability of this approach and our results into question. Again, this study constitutes a necessary stepping stone towards a practical solution for assessing incompleteness of contracts. First, there exist limited efforts in the research community to apply mutation analyses to contract-based software. To the best of our knowledge, there did not exist an available baseline before to compare against. Second, our experimental subjects cover typical implementations from single algorithms and smaller examples to design patterns, which can be found in most object-oriented languages. We are therefore confident that our results can be transferred to similar programming and specification languages. Third, our starting position are already verified implementations. Only few open-source projects exist in the literature that are reasonable in complexity and automatically verifiable with KEY. Therefore, many other projects had to be excluded. Finally, only the limited number of projects allowed us to perform the manual assessment needed for establishing the ground truth.

In our study, we did not focus on scalability of our mutation analysis and did not report any performance metrics. The reason is that our setup is only of prototypical nature. We additionally needed manual effort in parts of our evaluation. Moreover, the limited number of generated mutants could be verified reasonably fast, but verification effort in general may depend on many factors (e.g., employed verifier, configurations, or complexity of implementations). As we rather focused on the effectiveness of a mutation analysis for contract-based software and not so much on the evaluation strategy, we acknowledge that performance may play a crucial role for developers and should be addressed in large-scale evaluations in the future.

## 5.5. Related Work

There exists a rich corpus of impactful research focusing on mutation testing (i.e., mutation analysis for software testing) [Mathur 2013; DeMillo et al. 1978; Budd et al. 1980; Mathur et al. 1994; Offutt et al. 1996; Daran et al. 1996; Andrews et al. 2005; Just et al. 2014]. In the following, we discuss related work close to the topic of this chapter, namely mutation analysis for software contracts, mutation analysis in the broader context of formal verification, and measurement of incomplete specifications in general.

### Mutation Analysis for Software Contracts

As far as we know, Le Traon et al. [2006] were the first who successfully applied a mutation analysis to software contracts to evaluate their completeness, which they termed *contract efficiency*. They targeted behavioral specifications in Eiffel, where they mutated implementations to create faulty versions of their programs under analysis. Moreover, they report on two additional software quality measures that can be of interest for developers working with software contracts. First, *vigilance* is a metric that measures the ability of a program to identify faulty implementations at runtime. Since contracts aim at specifying safe behaviors, they can serve as test oracles and be evaluated at

runtime. Vigilance can therefore be used to further investigate contract completeness. Second, *diagnosability* measures the effort needed to locate defects. The work of this chapter differs in several aspects. First, we apply deductive verification to verify mutants instead of generating test cases that are executed at runtime. Test case generation has the problem that only a sample of inputs is tested, whereas formal verification aims at checking all input. It is certainly interesting to evaluate several more *evaluation strategies* for mutants, such as test case generation and model checking, in terms of performance and effectiveness in the future. Second, we conducted an empirical study on the effectiveness of mutation analysis for software contracts and contribute the first baseline for JML projects, whereas the work by Le Traon et al. [2006] targeted Eiffel specifications and merely applied a simplified mutation analysis to have some notion of contract completeness to compare against. Finally, we focus primarily on improving the strength of software contracts and not on identifying defects in implementations.

Du Bousquet et al. [2010] also highlighted that mutation analyses and the resulting mutation score may be used to increase trust in software contracts. They base their work on JML and the WHY platform, but only outlined the idea and did neither present a full solution nor an empirical evaluation. In particular, they illustrated their idea on a contrived toy example concerned with computing the maximum number between two integers, but did not define the mutation analysis in detail. The work of this chapter is a direct continuation of their theoretical considerations and reports on a full solution for mutation analysis of JML projects, including the introduction of contract-level mutations and an extensive evaluation. Similar to our goal, Groce et al. [2018] aim at investigating to what extent verification results reflect the developers intention. They combine mutation testing with model checking for C code, and use dedicated assertion macros as specification language.

While standard source-code mutation operators for JAVA were applied to mutate the implementations [Ma et al. 2006], contract-level mutation operators were adopted from Hou et al. [2007]. Their goal was to mutate interface specifications of black-box components to measure test adequacy of their black-box test suite. They proposed mutation rules for weakening preconditions and strengthening postconditions, as explained in Section 5.3.2. In our white-box setting, we evaluated whether the proposed mutation rules are effective to analyze a contract's strength. Our results let us hypothesize that strengthening postconditions following the proposed mutation rules is less fruitful than mutating the source code, whereas weakening the precondition gives raise to a new quality measure. Similarly, Ball et al. [2008] address coverage of system implementations by mutating specifications in combination with formal verification, which they label as *vacuous analysis*. In contrast to our approach, they do not follow the design-by-contract paradigm, but analyze temporal logic and automata in combination with model checking.

## Mutation Analysis in Formal Verification

Besides analyzing completeness of program specifications, mutation analysis was recently applied in the context of proof assistants dealing with higher-order logic. Celik et al. [2019] propose the tool chain mCoq, which performs a mutation analysis on Coq source files. They coined the term *mutation proving*, which aims at mutating functions and data types specified in Coq's functional programming language GALLINA. Similar to identifying incomplete contracts, alive mutants constitute vaguely specified definitions that are consequently less applicable and ex-

hibit lower programming quality overall. Laibinis et al. [2021] adopt mutation testing for evaluating adequacy of verification rules in the context railway signaling data. They argue that a large number of mathematical conjectures must be formulated, which puts validity and completeness of these conjectures into question. Their experiences and discussion emphasize that synthesizing mutants is both practical and promising, but also comes with the limitation that results greatly depend on the set of analyzed subjects.

Furthermore, mutation analysis is also applied to evaluate effectiveness of formal verification tools. Rao et al. [2017] proposed a novel process inspired by mutation testing to evaluate formal verification tools in the automotive and aerospace domains. In particular, they introduced known defects into Simulink specifications and evaluated to what extent developers are able to identify them.

## Specification Completeness in General

As far as we know, only little research has been published regarding coverage metrics in the context of formal verification. Polikarpova et al. [2013] discuss the role of strong specification in the programming language Eiffel and C#, and propose a methodology that supports the writing of stronger specifications for developers. In their extensive empirical evaluation, they highlighted that strengthening specifications led to the discovery of numerous bugs in their tested subjects, where specification were used as test oracles. This is in line with our motivation, where stronger specifications correspond to lower defects overall.

Chockler et al. [2003] adopted coverage metrics from simulation-based verification (i.e., syntactic and semantic coverage metrics), and re-formulated them in the context of model checking. Their version of *mutation coverage* corresponds with the challenge addressed in this chapter. Other relevant and discussed coverage metrics include *assertion coverage* (i.e., *does my execution satisfy the assertion?*) and *code coverage* (i.e., *which branches and statements are executed?*). Although their considerations are only of theoretical nature and target finite-state machines as underlying implementation language, evaluating the proposed coverage metrics in the context of contract-based projects or adopting model-checking for additional information are two interesting continuations of this line of research.

Alternatively to mutation analysis, Ghassabani et al. [2017] proposes to calculate coverage metrics based on *inductive validity cores* [Ghassabani et al. 2016]. Inductive validity cores constitute minimal sets about logical properties that establish a proof for particular verification problems. They argue that this approach is more cost-effective than mutation analysis, as inductive validity cores are efficient to compute, and employ in their work again model checking. Although Ghassabani et al. [2017] only report on initial results, it is interesting to compare the results we obtained in our evaluation to technique that are unrelated to a mutation analysis.

## 5.6. Chapter Summary

We addressed the challenge of measuring contract completeness by combining mutation analysis with deductive verification, whereas we focused primarily on Java/JML programs. Nonetheless, we are confident that the results we obtained are transferable to similar behavioral specification languages that follow the design-by-contract methodology. Besides 16 source-code mutation opera-

tors, we also incorporated two contract-level mutation operators and evaluated their usefulness. Our evaluation was conducted on ten open-source JML projects comprising a total of 77 analyzed methods. To the best of our knowledge, we provide the first prototypical open-source tool support that combines mutation analysis with contract-based software, which we base on  $\mu$ JAVA and KEY.

Although our evaluation was limited in size and complexity, we gained several insights that may shape the future of this line of research. First, incomplete software contracts occur frequently. This emphasizes the importance of an adequacy measurement for contract completeness, as uncovered parts of the implementation may very well mask minor bugs or even safety-critical defects. Second, the more complex method contract and implementation are, the more effective a mutation analysis seems to be, as more mutants are generated. In particular, incomplete contracts that were not identified by our analysis were mostly caused by too few generated mutants. Third, mutation rules for mutating the postcondition were rather ineffective, while mutation rules for mutating the precondition gave rise to a new metric that can support developers. Finally, the mutation score can give a good approximation for specification coverage of some projects (e.g., OpenJML), but a high mutation score may not always be corresponding to a very strong contract (e.g., DutchFlagAlgorithm). That is, every alive mutant can lead to important discoveries.

In the future, it will be necessary to reproduce this study on a larger and more complex corpus of experimental subjects, and also to generalize this approach to other programming and specification languages. Moreover, more sophisticated contract-level mutation operators based on (efficient) synthesis algorithms could further improve mutation of postconditions. Conversely, we may also consider other application areas than typical software engineering practices, such as evaluating the quality of *synthesized* contracts, or reporting the mutation score as an adequacy measurement for contract strength when open-source JML projects are used as benchmarks in other evaluations.

Measuring contract completeness is one important challenge for developers working with contract-based software. A different challenge also important in industrial settings is automation and performance with respect to formal verification of software contracts. We discuss this challenge tailored to *configurable* automatic program verifiers in the next chapter.





# 6. GUIDO: Guiding Developers in Configuring Deductive Program Verifiers

*This chapter shares material with the ITP'18 paper “Understanding Parameters of Deductive Verification: An Empirical Investigation of KeY” [Knüppel et al. 2018c] and the FormaliSE'21 paper “GUIDO: Automated Guidance for the Configuration of Deductive Program Verifiers” [Knüppel et al. 2021b].*

In the previous chapter, we discussed how a mutation analysis applied to contract-based software can support developers in their verification projects. Besides quantifying incompleteness of specifications, there are many other challenges at the intersection of software engineering and formal verification which hinder a successful adoption in industry. Two of the main arguments against such an adoption are insufficiency of automation (or lack thereof) and inadequate feedback in case of failed verification attempts. Even automated theorem provers fail frequently due to a multitude of reasons, which are typically difficult to understand for non-expert users. In this chapter, we address a subproblem in this context that we identified, namely *parameterization of configurable program verifiers*. Essentially, many automated formal verification tools allow users to adjust parameters that control different parts of the automatic proof search, while default configurations are oftentimes not optimal [Knüppel et al. 2018c].

While most prominent model checkers are configurable and provide many different techniques and heuristics to automate the configuration process (see CBMC [Kroening et al. 2014], CPACHECKER [Beyer et al. 2011], VERIABS [Darke et al. 2018], and PESCO [Richter et al. 2019]), automatic configuration for theorem provers in general and deductive program verifiers in particular, such as KEY [Ahrendt et al. 2016] or FRAMA-C [Cuoq et al. 2012], has gained only little attention in the research community so far. They require users to adjust parameters for their automatic proof search in case the default configuration is insufficient. However, finding a good configuration often requires significant expert knowledge of the underlying proof theory and the verification tool itself to understand the effect certain parameters may have [Knüppel et al. 2018c].

For the purpose of this chapter, we describe the adequacy of configurations in two dimensions. Primarily, a configuration should suffice to let the internal verification procedure check correctness of a program automatically, which we refer to as *verifiability*.<sup>1</sup> For instance, following the *design-by-contract* paradigm [Meyer 1992], verifiability signifies whether a method conforms to its contract. However, if verification fails even though the input program is correct, practitioners often tend to

---

<sup>1</sup>In the context of deductive verification, which focuses on finding proofs, *provability* is a synonymous term. We use the more general term *verifiability* to also include other verification disciplines with less focus on proofs, such as model checking.

refactor their implementation or specification, rather than tweaking the configuration [Runge et al. 2019b]. The second dimension is *verification effort*, which is a quantified measure of the verifier’s performance for a given task (e.g., execution time, memory consumption, or proof size).

Although verifiability is often the primary focus, there is an inevitable demand in practice to reduce the verification effort [Gleirscher et al. 2018]. Considering continuous integration, software systems evolve frequently, as new and modified source code is committed often. That means, verification tools need to reason continuously [O’Hearn 2018] about the correctness of software. Verifying a program once is therefore almost negligible compared to the many verification tasks performed when incorporating verification tools into continuous integration processes. In these cases, saving even a small percentage of verification effort may accumulate quickly over time, which makes formal verification more affordable.

In this chapter, we discuss how to automate the configuration selection process for configurable program verifiers in general, whereas our focus is on deductive verification and the program verifier KEY. In particular, we propose GUIDO, a holistic framework and tooling based on *statistical hypothesis testing*. That is, GUIDO allows domain experts (e.g., tool builders) to channel their knowledge and experience by formalizing hypotheses about the assumed effects on verifiability and verification effort of various configuration options. This knowledge is then accessible by normal users (e.g., developers) to identify adequate configurations automatically. We argue that incorporating explicit domain knowledge into this process overcomes many limitations that general-purpose tools for performance prediction of configurable software [Siegmond et al. 2012; Wu et al. 2015; Ha et al. 2019; Guo et al. 2018] have in this specific context. First, they are highly configurable themselves, which only shifts the problem of parameter optimization. Second, they are mainly regression-based and need to learn the influence of configuration options on their own. This requires significantly more training data, which is still challenging to obtain in the context of formal verification. We are confident that GUIDO can be valuable to (1) inexperienced practitioners for an easier entry to automated program verification, (2) experienced practitioners for improving productivity in their verification projects, and even (3) researchers and tool builders for evaluating verification techniques and the effects of newly introduced configuration options.

In Section 6.1, we outline the challenges of configurable program verifiers using KEY as example. Next, we present a high-level overview of GUIDO in Section 6.2 and explain the core technical aspects of GUIDO in Section 6.3. In Section 6.4, we elaborate on GUIDO’s open-source implementation, and in Section 6.5, we discuss the application of GUIDO to KEY-2.7.0. We evaluate GUIDO empirically in Section 6.6. While our main focus is on deductive verification, where we thoroughly investigate KEY-2.7.0, we also apply GUIDO in the model checking domain to CPACHECKER-1.8 [Beyer et al. 2011] to evaluate generalizability of our approach. Finally, we discuss related work on configuration prediction and configurable program verifiers in Section 6.7.

## 6.1. Problem Statement

Formal verification complements software testing by identifying the last remaining defects that are otherwise difficult to find. In particular for contract-based software and deductive verification, practitioners are confronted with numerous potential causes if automatic verification fails, where the exact cause is oftentimes tedious to identify [Knüppel et al. 2018a; Knüppel et al. 2018c]. For

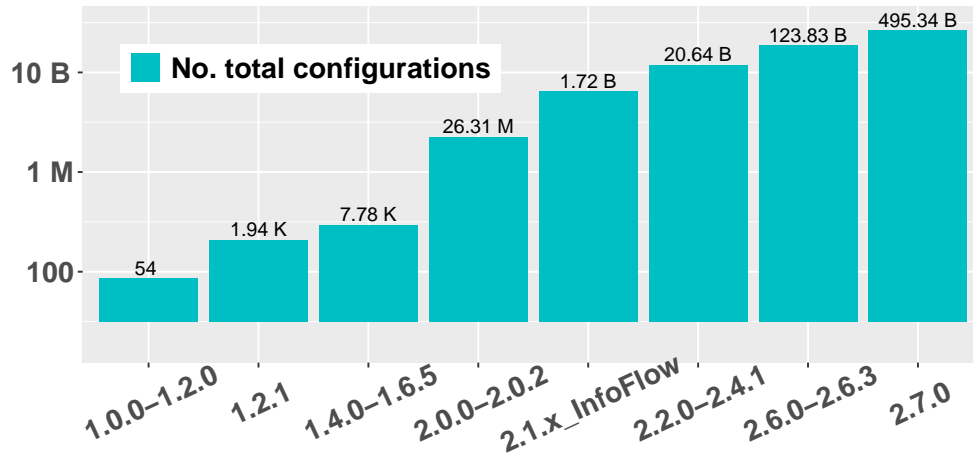


Figure 6.1.: Number of applicable configurations in logarithmic scale for the program verifier KEY [Ahrendt et al. 2016] from version 1.0.0 to version 2.7.0 (adopted from [Knüppel et al. 2021b]).

instance, three common reasons for failed proof attempts are defective implementations, insufficient specifications, or insufficient tool support (e.g., when the automatic proof search is not powerful enough). Inexperienced practitioners may not directly identify the exact cause, as feedback in case of failure for theorem proving is mostly vague (see Section 2.1.2). A related challenge addressed in this chapter focuses on *configurable* program verifiers. Essentially, *parameterization* of the underlying proof search algorithm of such program verifiers can have a great effect on their automation and performance, and default settings are many times insufficient as well [Knüppel et al. 2018c]. One goal for practitioners is therefore to identify the right configuration for a particular verification problem that optimizes verifiability and verification effort.

The number of configuration options of various verification tools may also increase over time with each new release due to the nature of software evolution. At the same time, each additional configuration option also makes the overall configuration space larger, even leading to a combinatorial explosion in the number of applicable configurations. End-users and even experts may struggle to understand the influence certain parameters have, and applying all of them becomes impractical due to large configuration spaces. As an example, we depict the number of applicable configurations for the program verifier KEY from version 1.0.0 up to version 2.7.0 in Figure 6.1. Whereas version 1.0.0 started with eleven configuration options spread over four control parameters, version 2.7.0 consists of 56 configuration options spread over 30 control parameters, leading to more than 495 billion applicable configurations.<sup>2</sup>

We distinguish between two types of parameters. First, *qualitative* parameters affect certain properties subject to verification by asking the question: *what is verified?* For instance, KEY provides the option to ignore integer overflows during verification instead of considering JAVA semantics, which allows successful verification of some essentially defective JAVA programs. The second type describes parameters that directly influence the proof search algorithm (i.e., algorithmic configuration) with respect to success rate or performance (i.e., *how is it verified?*). For instance, KEY

<sup>2</sup>In the remainder of this chapter, we will use the term *parameter* to mean the dedicated change to a part of the proof search algorithm offered by the user interface (e.g., *method call treatment*) and *configuration option* to refer to a specific value of a parameter (e.g., *contracting* or *inlining*).

---

<pre> 1  /*@ public normal_behavior 2    @ requires T &gt; 0; 3    @ ensures \result &lt; T; 4    @*/ 5  public /*@ pure @*/ int modT(int input, int T) { 6    return input % T; 7  } </pre>	Computing Modulo
--	------------------

---

<pre> 8  /*@ public normal_behavior 9    @ requires e != null; 10   @ ensures contains(e); 11   @ ensures collectionSize == \old(collectionSize) + 1; 12   @ ensures \result; 13   @ assignable elements; 14   @*/ 15  boolean add(/*@ nullable @*/ Object e) { 16    ... 17  } </pre>	Class ArrayList
--	-----------------

---

Listing 6.1: Two methods specified with contracts in JML (adopted from [Knüppel et al. 2018c]).

provides a parameter to replace method calls with their respective contract instead of inlining the implementation. In many cases, proof obligations based on contracts are easier to discharge, which will lead to an improvement in performance.

While qualitative parameters must be identified by end-users before starting the verification process, non-expert users typically resort to the default configuration for the underlying proof search algorithm. If verification fails (e.g., if the verifier runs into timeout) or performance digresses from expectations, tweaking the control parameters is typically the last considered resolution. The reason is that the influence of configuration options is oftentimes not well understood, as most modern verifiers are rather driven by research than by industry, where both areas differ in budget, time, and focus. Although we assume that the default configuration is a good starting point, we show in [Example 6.1](#) that the default configuration is sometimes insufficient.

**Example 6.1.** In [Listing 6.1](#), we illustrate two examples in JML, where the default configuration is either insufficient or results in higher verification effort than necessary. The top method `modT(int input, int T)` consists of two input arguments (i.e., dividend `input` and divisor `T`) and computes the remainder between them based on the modulo operation. The postcondition only states that the remainder is always less than the divisor. In KEY-2.6.1, this method was not automatically verifiable due to the default configuration. In particular, KEY’s parameter *Arithmetic treatment* must be set to one of three configuration options, namely *Basic*, *DefOps*, or *Model Search*. The default option *Basic* is insufficient for evaluating the modulo operator, whereas option *DefOps* suffices.<sup>3</sup>

---

<sup>3</sup>Due to some notion of syntactical equivalence, changing the postcondition to `\result == input % T` would also suffice to verify method `modT(int input, int T)`. However, although incomplete, the software contract as-is still conforms to the implementation.

The second method is part of class `ArrayList` of the `Collection-API`, which we specified and explained in more detail in previous work [Knüppel et al. 2018a]. We illustrate an excerpt of the software contract for method `add(Object e)`. Although the illustrated software contract is relatively simple, the size of the resulting proof from KEY’s automatic proof search can be reduced by over 60% (i.e., from 33,748 proof steps to 13,328 proof steps) when parameter `Quantifier treatment` is set to `No Splits` instead of `No Splits with Progs`.

The example indicates that identifying the right configuration can be difficult, but seems necessary to increase successful application of verification tools. In the next section, we give an overview of our contribution, GUIDO, which aims at automatically deriving adequate configurations for program verifiers while considering the impact on verifiability and verification effort.

## 6.2. Overview of GUIDO

Before we lay out the technical contributions of our approach in the next section, we first define what we mean by *configurable verification system* in Section 6.2.1. Next, we motivate the definition of explicit domain knowledge in form of hypotheses in Section 6.2.2 and present a high-level overview of GUIDO in Section 6.2.3.

### 6.2.1. Configurable Verification Systems

The field of application for GUIDO are configurable deductive verification systems that let a user adjust the automatic verification process. For the sake of presentation, we only focus on binary configuration options (i.e., 1 if a configuration option is selected and 0 otherwise). We denote by  $\mathcal{O}_S$  the set of all possible configuration options of a particular verification system  $S$ . As not all combinations of options are valid (i.e., configuration options may be alternative to each other), there exist constraints among them. We implicitly encode such constraints by defining the set of all valid configurations of  $S$  as  $\mathcal{C}_S$ . In practice, the space of configurations is symbolically represented by a propositional formula in conjunctive normal form, and can be analyzed using satisfiability (SAT) solvers [Ahmed et al. 2017; Mendonca et al. 2009]. Then, a particular configuration  $c \in \mathcal{C}_S$  is modeled by a function  $c : \mathcal{O}_S \rightarrow \{0, 1\}$  that maps each configuration option  $o \in \mathcal{O}_S$  to either selected (1) or deselected (0).

A frequently applied constraint among a set of configuration options is *mutual exclusion*, which means that *exactly one* of these options has to be selected. Hence, we assume that configuration options are *grouped*, and all configuration options in the same group are mutually exclusive. To use common terminology, we refer to these groups as *parameters*, which means that a parameter can be set to a specific configuration option. For instance, when we want to verify a program, there may exist one parameter consisting of two configuration options to control how integers are handled: (a) either as purely mathematical objects with infinite domains or (b) with the semantics of the used programming language. A parameter represents the abstract intention (e.g., *how integers are handled*), whereas the contained configuration options represent the actual setting. We denote by  $\mathcal{P}_S \subseteq 2^{\mathcal{O}_S} \setminus \{\emptyset\}$  the set of parameters for a given verification system  $S$  and require that all parameters in  $\mathcal{P}_S$  are pair-wise disjoint, such that any configuration option is only element of exactly one parameter (i.e.,  $\forall p_1, p_2 \in \mathcal{P}_S \wedge p_1 \neq p_2 : p_1 \cap p_2 = \emptyset$ ).

**Example 6.2.** The program verifier KEY [Ahrendt et al. 2016] provides parameter One Step Simplification, which may allow the automatic proof search to reduce multiple proof steps into a single one. Formally, we define parameter One Step Simplification as  $p_{\text{oss}} = \{o_d^{\text{oss}}, o_e^{\text{oss}}\}$  with two configuration options: either Disabled ( $o_d^{\text{oss}} = 1 \wedge o_e^{\text{oss}} = 0$ ) or Enabled ( $o_d^{\text{oss}} = 0 \wedge o_e^{\text{oss}} = 1$ ). Both configuration options are mutually exclusive, such that exactly one of them has to be selected (i.e., for every configuration  $c \in C_S$ ,  $c(o_d^{\text{oss}}) \neq c(o_e^{\text{oss}})$  holds).

## 6.2.2. Statistical Hypothesis Testing

The research goal of this chapter is to build a practical framework that supports developers in deriving adequate configurations for program verifiers automatically. Although this goal is related to the field of performance prediction for configurable software [Siegmund et al. 2015], we argue that state-of-the-art tools based on regression analysis in that area [Siegmund et al. 2012; Ha et al. 2019; Guo et al. 2018] are too general to be applicable in the context of deductive verification. First, we assume that the verification effort (e.g., execution time) for program verifiers is not normally distributed, which is a fair assumption for configurable software in general. Second, most of these tools work in a black-box manner (i.e., measuring execution time of the compiled version), whereas verifiability and verification effort greatly depend on the structure of implementation and specification. Third, collecting enough data to make predictions is difficult in the context of formal verification due to low adoption rates and general scalability problems. These tools, however, need a large corpus of data to minimize the prediction error.

To address these problems of general-purpose tools, we propose (1) to apply a *white-box* analysis and (2) to incorporate *domain knowledge* about the influence of parameters in form of statistical hypotheses. The benefit of formalizing hypotheses is threefold. First, instead of letting a prediction algorithm identify such hypotheses on its own by analyzing lots of data (i.e., as performed in regression analyses), encoding knowledge about the influence of parameters should require less data and be more precise. Second, such a formal foundation of encoding domain knowledge generalizes to other formal verification tools and even to other disciplines (e.g., compilers with its optimization parameters). Third, statistical hypotheses can be tested, which means that false domain knowledge is identifiable. In Example 6.3, we illustrate how hypotheses about configuration options can be expressed.

**Example 6.3.** We revisit Example 6.2, where we formalized parameter One Step Simplification. Corresponding hypotheses focusing on either verifiability or verification effort can take the following form.

- **Hypothesis (Verifiability):** If a specification case is verifiable with option  $o_e^{\text{oss}}$ , it is also verifiable with option  $o_d^{\text{oss}}$  (i.e.,  $o_d^{\text{oss}}$  is at least as effective).
- **Hypothesis (Verification effort):** If the verification task's implementation contains loops, the verification effort with option  $o_d^{\text{oss}}$  is at least as large as with option  $o_e^{\text{oss}}$  (i.e.,  $o_e^{\text{oss}}$  is at least as efficient for loop-containing programs).

Whereas the first hypothesis on verifiability ignores language constructs, the second hypothesis on verification effort only applies in the presence of loops.

Ideally, the right configuration maximizes verifiability while it also minimizes verification effort. However, in previous work, we found evidence that both criteria are on opposite sites of a



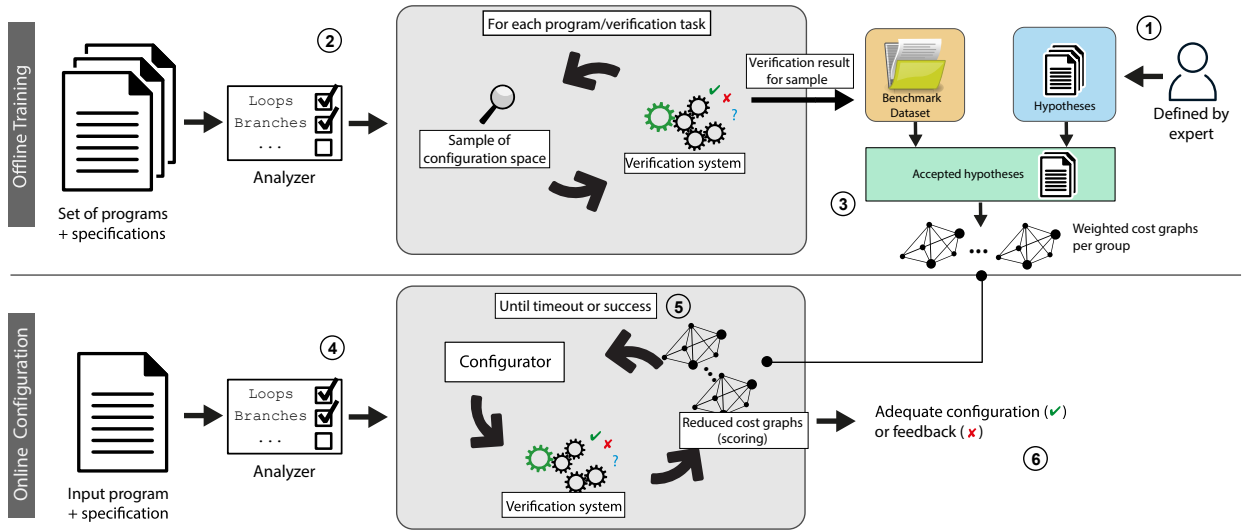


Figure 6.2.: Schematic workflow of GUIDO’s offline training phase and online configuration search to compute adequate configurations automatically. By scoring options of the verification system on formulated hypotheses and the benchmark dataset, GUIDO determines a sequence of ranked configurations for a given verification task. End-users can optimize for either verifiability or verification effort (adopted from [Knüppel et al. 2021b]).

continuum for some configuration options [Knüppel et al. 2018c] (i.e., a configuration option typically only improves one of the two criteria at the same time).

### 6.2.3. GUIDO Workflow

Based on the set of user-defined hypotheses, our idea is to develop a holistic framework that paves the way for an *automatic search* for configurations with reasonable trade-off, which we call GUIDO. The goal of GUIDO is to support practitioners in formulating hypotheses, data collection, and configuration prediction.

In Figure 6.2, we present a high-level overview of GUIDO, which is divided into an *offline* training phase and an *online* phase for the configuration prediction. The focal point of the offline training phase is to learn a *cost model* of the verification system’s parameter space that can then be used in the online phase. Although costly, this phase has to be performed only once for each particular release of interest. The offline phase starts with formalization of the domain knowledge ①. That is, a user defines hypotheses about the influence of configuration options with respect to verifiability and verification effort as described above. Ideally, hypotheses are directly formulated by domain experts, but may also be *explored* by reading tool tips, the documentation, or publications [Knüppel et al. 2018c]. Next, a *benchmark dataset* is established by verifying numerous training tasks with a representative sample of configurations and gathering information about each verification attempt (e.g., success, verification effort, or syntactical structure of the tasks) ②. Finally, all hypotheses are tested with respect to the benchmark dataset ③. Accepted hypotheses are used to compute weighted cost graphs that formally reflect the influence of certain configuration options.

The online configuration search is applied to each input verification task, for which a fitting configuration must be identified. Again, the input verification task is analyzed to gather information about its syntactical structure ④. Next, GUIDO uses the weighted cost graphs to score each configuration option individually with respect to the input task's characteristics and a user's focus (i.e., verifiability or verification effort). Based on this scoring, a *linear program* (i.e., constraint optimization problem) is formulated and solved to determine the optimal configuration ⑤. If the verification attempt fails with the optimal configuration, a *continuation mechanism* is applied to compute a different but also promising configuration. A continuation mechanism helps to address overfitting, where the benchmark dataset does not reliably reflect characteristics of the new input task. Additionally, a user-defined timeout caps the maximum time that GUIDO spends on the configuration search and verification of the input task. Finally, either GUIDO identifies a useful configuration or provides a list of tried configurations for manual inspection ⑥.

The benefits of a framework such as GUIDO are manifold. First, many verification tools already exist that could benefit right now and lower the entry barrier for practitioners (i.e., both, experts and non-experts). Second, in times of continuous integration and reasoning, GUIDO alleviates unnecessary labor, as fitting configurations are found automatically instead of finding them manually. For instance, if a good configuration for a specific part of the software is found, chances are that this configuration is also a good fit for future verification. Third, particularly for deductive verifiers that generate *proofs*, a goal is not only to successfully verify a program, but also to find smaller proofs (i.e., to be resource-beneficial in case of distribution [Necula 1997] or proof replay [Beckert et al. 2004; Bubel et al. 2016]). After a successful verification attempt, GUIDO can try more promising configurations to optimize proofs. Fourth, in case of dependable software systems, it often becomes necessary to manually switch configurations during a verification task (e.g., when the automated proof search stops and does not know how to proceed), which is hardly addressed by any modern configurable verification tool. Fifth, each software project is different, which emphasizes the benefits of a learning-based approach that is able to adapt to a diverse set of verification tools.

## 6.3. A Data-Driven Framework for Automatic Configuration

In this section, we introduce the core technical contributions of GUIDO, which are divided into the offline training phase and the online configuration search. In Section 6.3.1, we first focus on the offline training phase by explaining how the benchmark dataset is acquired and how user-defined hypotheses are tested against it. Next, in Section 6.3.2, we discuss how costs of individual options are computed and how the online configuration search is formulated as a linear optimization problem.

### 6.3.1. Offline Training: Data Set Acquisition and Hypothesis Testing

#### Verification Procedure and Benchmark Dataset

GUIDO aims at being agnostic to the underlying verification system, which is why we introduce some abstract terminology to formalize its core techniques as general as needed. For a defined verification system  $S$ , we use the abstract term *verification task* to refer to programs (or parts thereof) that are subject to verification. For deductive verification, these are typically specification cases

of methods (see [Section 2.1.2](#)). We denote by  $\mathcal{T}$  the universe of all verification tasks of interest. Furthermore, we assume that verifying a verification task will produce a *verification result* from a universe  $\mathcal{R}$  of such verification results, which we specify further below. First, we define an abstract version of a verification procedure as follows.

**Definition 6.1: Verification Procedure**

Let  $T \in \mathcal{T}$  be a verification task and  $c \in \mathcal{C}_S$  be a configuration for verification system  $S$ . A **verification procedure** has the following signature:

$$\mathcal{V} : \mathcal{T} \times \mathcal{C}_S \rightarrow \mathcal{R}.$$

Applying  $R = \mathcal{V}(T, c)$  on a verification task  $T \in \mathcal{T}$  with configuration  $c \in \mathcal{C}_S$  will produce a verification result  $R \in \mathcal{R}$ , which consists of three parts: (1) whether the input verification task was verified, (2) the real-valued verification effort, and (3) a set of identified language constructs of the input verification task. As described in the previous section, GUIDO identifies syntactical language constructs of the input verification task (e.g., loops, branching, or parts of a specification), which are used as *features* to guide the configuration search. Set  $\mathcal{L}$  denotes all identifiable language constructs, where  $L \subseteq \mathcal{L}$  represents a subset of source code and specification constructs identified by a static lightweight analysis. Formally, we define the structure of a verification result as follows.

**Definition 6.2: Verification Result**

A **verification result**  $R \in \mathcal{R}$  is a tuple  $(r_{\text{ver}}, r_{\text{eff}}, L)$ , where:

- $r_{\text{ver}} \in \{0, 1\}$  states whether the program could be verified successfully (i.e., 1) or not (i.e., 0),
- $r_{\text{eff}} \in \mathbb{R}$  is the measured verification effort (e.g., execution time, proof size, or memory consumption),
- and  $L \subseteq \mathcal{L}$  is the subset of identified source code and specification constructs.

Incorporating the identification of relevant language constructs aims at improving the quality of the automated configuration search, as the structure of a program has a great influence on the verification result. The set of relevant language constructs  $\mathcal{L}$  is typically provided by an expert and depends on the verification domain (e.g., employed program verifier and programming language). Typical constructs appear as boolean flags (e.g., occurrences of *loops* or *branches*), but may also be parameterized (e.g., more than  $n$  lines of code). In [Table 6.1](#), we show prominent language constructs. For instance, if the analyzed program contains a while-loop, then  $\text{loop} \in L$ .

To establish a *benchmark dataset* of verification results, GUIDO needs a set of *benchmark verification tasks*  $\mathbf{T}_{\text{be}} = \{T_1, \dots, T_n\}$  and a set of valid prover configurations  $C \subseteq \mathcal{C}_S$ . For deductive verification [[Ahrendt et al. 2016](#)], we consider a verification task to be exactly one of (possibly) numerous specification cases of an object's method (e.g., as promoted by using software contracts [[Meyer 1992](#)]). Evaluating tasks in  $\mathbf{T}_{\text{be}}$  with every configuration in  $\mathcal{C}_S$  is typically infeasible when the configuration space  $\mathcal{C}_S$  is too large. We apply a common solution to this problem, namely *t-wise* sampling [[Johansen et al. 2012](#); [Varshosaz et al. 2018](#); [Garvin et al. 2011](#)], which ensures that all possi-

	Construct	Description
Specification	quantifiers	Uses quantification (i.e., $\forall$ or $\exists$ )
	implications	Uses implication (i.e., $\Rightarrow$ or $\Leftrightarrow$ )
	frame	Assumes unmodifiable locations
	precondition	Precondition different from <code>true</code>
	postcondition	Postcondition different from <code>true</code>
	<code>los(<math>n</math>)</code>	More than $n$ lines of spec.
Source Code	branching	Contains <code>if</code> -statements
	loops	Contains loops
	arrays	Uses arrays
	recursive	Is recursive
	arguments	Gets arguments as input
	<code>returnValue</code>	Has a return value
	<code>callDepth(<math>n</math>)</code>	Call depth is greater than $n$
	<code>sloc(<math>n</math>)</code>	More than $n$ lines of code
	<code>methodCalls</code>	Calls other methods

Table 6.1.: Set of language constructs  $\mathcal{L}$  identified by a source code and specification analysis.

ble combinations of  $t$  configuration options are covered at least once if the set of valid configurations  $\mathcal{C}_S$  allows it. Even for small values of  $t$ , such as 2 or 3,  $t$ -wise sampling is known to produce a good trade-off between small and representative subsets of all valid configurations [Johansen et al. 2012]. We define the benchmark dataset as follows.

#### Definition 6.3: Benchmark Dataset

Let  $\mathcal{V}$  be a verification procedure for verification system  $S$ ,  $\mathbf{T}_{be} \subseteq \mathcal{T}$  a set of benchmark verification tasks, and  $\mathcal{C}_S$  a sampled set of configurations. The **benchmark dataset**  $\mathcal{D}_S \in 2^{\mathcal{R}}$  is defined as

$$\mathcal{D}_S = \{\mathcal{V}(T, c) \in \mathcal{R} \mid T \in \mathbf{T}_{be} \wedge c \in \mathcal{C}_S\}. \quad (6.1)$$

Based on **Definition 6.3**, the benchmark dataset is established by verifying each training verification task  $T \in \mathbf{T}_{be}$  with each sample configuration  $c \in \mathcal{C}_S$ , and then storing the verification result in  $\mathcal{D}_S$ .

### Statistical Hypothesis Testing

As explained in the previous section, GUIDO is built on the idea of explicitly integrating formalized domain knowledge to improve the configuration search. That is, experts formulate *assumptions* about whether specific configuration options have a significantly greater influence than others for a given class of verification tasks. However, even experts may unintentionally introduce false or irrelevant assumptions and consequently compromise GUIDO’s effectiveness. To prevent the introduction of ill-posed domain knowledge, GUIDO ignores irrelevant hypotheses by applying standard

Criterion	Condition	Applied Test	Reference
<b>Verification Effort</b>	$p = \{o_1, o_2\}$	1-Sample Wilcoxon test	[Wohlin et al. 2012]
	$ p  > 2$	Paired Wilcoxon test	
<b>Verifiability</b>		McNemar test + inspection via contingency tables	[McNemar 1947]

Table 6.2.: Applied significance tests for a hypothesis  $H_{\text{ver/eff}} = (o_1, o_2, L)$  over a parameter  $p \in \mathcal{P}_S$ .

hypothesis testing to objectively validate all formulated assumptions beforehand. Formally, we denote by  $\mathcal{H}_S$  the set of hypotheses defined by experts and define a *hypothesis* as follows.

#### Definition 6.4: Hypothesis

Let  $o_1, o_2 \in \mathcal{O}_S$  be two configuration options of the same parameter, and let  $L \subseteq \mathcal{L}$  be a (possibly empty) set of language constructs. A **hypothesis**  $H_{\text{ver/eff}} \in \mathcal{H}_S$  with  $H_{\text{ver/eff}} = (o_1, o_2) \times L$  represents the assumption that choosing option  $o_1$  over option  $o_2$  performs *better* with respect to the *verification criterion* (i.e., either verifiability ver or verification effort eff) and the identified language constructs in  $L$ .

For example, specific options may be more relevant in the occurrence of loops, branches, or a combination thereof, and not so relevant otherwise. Although particular options may have synergistic effects (e.g., selecting two options of different parameters outperform a third option), finding such interactions is typically non-trivial [Calder et al. 2003] and might even unnecessarily complicate the formulation of hypotheses. For the sake of simplicity, we assume that GUIDO currently only allows to formulate hypotheses that compare two configuration options of the same parameter.

As commonly practiced in statistics, we apply standard tests from the domain of frequentist statistics [Neyman 1977] (i.e., null hypothesis statistical testing) to validate our hypotheses. As a result, we compute the *p-value* for each tested hypothesis. Assuming that the hypothesis stating the opposite (i.e., null hypothesis) is correct, the p-value represents the probability that new observations will lead to results that are as extreme as the results we already observed. A smaller p-value signifies stronger evidence in favor of our actual hypothesis.<sup>4</sup>

For hypotheses about verifiability, we chose a *McNemar test* including an inspection of the significance via contingency tables [McNemar 1947]. For hypotheses about the verification effort, we chose a non-parametric Wilcoxon test [Wohlin et al. 2012]. The reason against the most frequently applied *t-test* [Wohlin et al. 2012] is that we cannot assume a normal distribution when measuring the verification effort. Although the t-test is said to be robust against deviation from a normal distribution with large sample sizes [Zimmerman 1998], using a non-parametric test is more reasonable in our case, as (1) we do not make assumptions about the size of our sample and (2) a too drastic deviation from the normal distribution is possible.

In Table 6.2, we list the applied significance tests for a given verification criterion and a condition on the compared configuration options. In particular, hypotheses are only reasonable for param-

<sup>4</sup>It is important to note that the p-value mainly signifies presence of statistical significance, but is typically not an adequate measure of magnitude to quantify the significance. It is therefore good practice to also report on the *effect size* [Cohen 2013] of a hypothesis as a measure to quantify the significance.

eters that comprise *at least* two alternative configuration options. For instance, checkboxes, which only allow to select or deselect a configuration option, are encoded as parameters with exactly two configuration options, namely true and false. For hypotheses about verification effort, where the respective parameter consists of exactly two configuration options, we apply a *1-sample* Wilcoxon test instead of a paired Wilcoxon test. For hypotheses about verifiability, we always apply the McNemar test. To prevent the increasing probability of *Type-I* errors (i.e., accepting invalid hypotheses) in our tests, we apply the *Bonferroni correction* [Shaffer 1995] instead of a fixed significance level of  $\alpha$ . Such a correction lowers the significance level depending on the number of experiments that are performed to mitigate the influence of biased data.

### Cost Graph Construction

To decide which configuration options of a parameter should be prioritized, we design a *cost graph* per parameter in  $\mathcal{P}_S$  that reflects our prior established domain knowledge (i.e., the accepted hypotheses). We give the following definition.

#### Definition 6.5: Cost Graph

A **cost graph**  $\mathcal{G}_p$  for parameter  $p \in \mathcal{P}_S$  is a weighted and directed multi-edge graph  $(V, E, \omega)$ , where:

- $V$  is a set of vertices and each  $v \in V$  represents exactly one configuration option in  $p$ ,
- $E$  is a set of directed edges,
- $\omega : E \rightarrow \mathbb{R}$  is a function associating each edge in  $E$  with a cost.

Intuitively, a cost graph illustrates which configuration options are more likely to improve or deteriorate the verification result with respect to the verification criterion. Each edge is associated with a particular hypothesis and the edge's weight represents the measure of magnitude of this hypothesis. To quantify the significance of a hypothesis, we use Pearson's correlation coefficient as effect size [Benesty et al. 2009], where a value of 0.1 indicates a small effect, 0.3 indicates a medium effect, and 0.5 indicates a large effect. To explain our idea, consider the visualized example of a cost graph in Figure 6.3 of a parameter with three vertices comprising three mutually exclusive configuration options  $o_1, o_2$ , and  $o_3$ . Here, we accepted three hypotheses. The upper two edges and their corresponding hypotheses state that selecting configuration option  $o_2$  instead of  $o_1$  and  $o_3$  reduces the verification effort. The third hypothesis states that, in the presence of loops, selecting configuration option  $o_2$  deteriorates verifiability compared to selecting option  $o_3$ . All hypotheses are represented in the cost graph by connecting the configuration options with three weighted and directed edges. Cost graphs are practically easy to compute, but also provide an intuitive understanding of the dependencies between options.

The meaning of edges in a cost graph is formally described as follows. Let  $o_1, o_2 \in \mathcal{O}_S$  be two configuration options of the same parameter  $p \in \mathcal{P}_S$ . Moreover, let  $H = (o_1, o_2, L)$  be an *accepted* hypothesis with  $L \in \mathcal{L}$ . Then there exists an  $e \in E$  such that  $e = (o_1, o_2)$  and  $\omega(e) = r$  if and only if  $\Pr(\mathcal{D}_S | \bar{H}) \leq \alpha$ . Again,  $r$  is Pearson's correlation coefficient resulting from testing (i.e.,



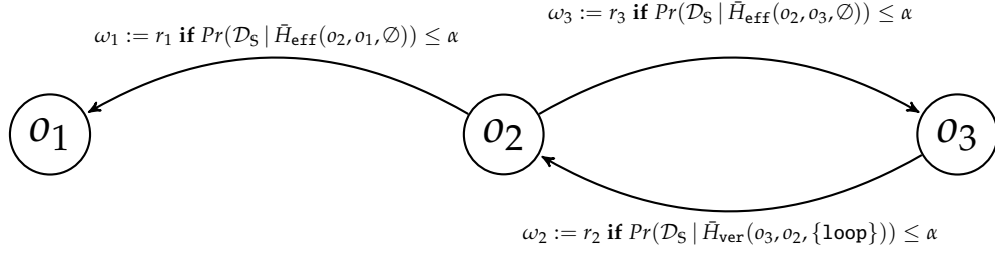


Figure 6.3.: Exemplified cost graph representing three accepted hypotheses. A directed edge indicates the acceptance of a user-defined hypothesis (i.e., considering the corresponding *p-value* resulting from function  $Pr(\cdot)$ ) and is weighted by Pearson’s correlation coefficient  $r_i$  [Benesty et al. 2009] with  $i \in \{1, 2, 3\}$  (i.e., using effect size as measure for significance).

rejecting) the null hypothesis  $\bar{H}$  based on benchmark dataset  $\mathcal{D}_S$ ,  $\alpha$  is the significance level (e.g.,  $\alpha = 5\%$  as commonly practiced in statistics [McKillup 2011]), and  $Pr(\cdot)$  denotes the measured *p-value* based on the chosen test and our benchmark dataset.

The *p-value* represents the probability that dataset  $\mathcal{D}_S$  reflects the *opposite* of what we initially stated with hypothesis  $H$  (i.e., null hypothesis is denoted by  $\bar{H}$ ). After a hypothesis is accepted, we use the effect size as a measure for its significance. The higher the effect size, the higher is the impact of the configuration option associated with  $H$ , which allows us to use the effect size as an adequate metric.

### 6.3.2. Online Configuration Search as an Optimization Problem

#### Cost Computation of Individual Configuration Options

In the offline phase, GUIDO uses all user-defined hypotheses to establish the benchmark dataset and to construct a cost graph for each parameter of interest. In the online configuration search, GUIDO aims now at using the cost graphs to *optimize* the configuration for an input verification task. Formally, this means that the established cost graphs  $\mathcal{G}_p = (V_p, E_p, \omega_p)$  per parameter  $p \in P_S$  together with the set of language and specification constructs  $L_T$  for an input task  $T$  are used to score each individual configuration option with essentially two score functions  $\theta_{\text{ver/eff}} : \mathcal{O}_S \rightarrow \mathbb{R}$ , one for verifiability and one for verification effort. To only take hypotheses into account that are formulated over a subset of the computed language constructs, we define  $\mathbb{1}_{L_T} : E \rightarrow \{0, 1\}$  as a function yielding 1 if  $L_H$  of the associated hypothesis of the input edge is a subset of  $L_T$  and 0 otherwise. The score functions are defined in the following way.

$$\forall o \in \mathcal{O}_S : \theta_{\text{ver/eff}}(o) = - \left[ \sum_{\substack{e \in E \\ \wedge e = (o', o)}} \mathbb{1}_{L_T}(e) \omega(e) \right] + \left[ \sum_{\substack{e \in E \\ \wedge e = (o, o')}} \mathbb{1}_{L_T}(e) \omega(e) \right] \quad (6.2)$$

The idea is that starting from a cost value of 0 for a configuration option (e.g., no hypotheses are associated with that option), outgoing edges increase the score whereas incoming edges decrease the score. In particular, we compute two different scores for each existing configuration option



---

**Algorithm 6.1:**  $\text{COMPUTEScore}_{\text{ver/eff}}(o, G_p, L_T)$ , where  $o \in \mathcal{O}_S$  is a configuration option of parameter  $p \in \mathcal{P}_S$ ,  $G_p = (V_p, E_p, \omega_p)$  is the cost graph for parameter  $p$ , and  $L_T$  is the set of identified language constructs for verification task  $T$ .

---

- 1: Let  $H_e = (o_1, o_2, L_e)$  be the hypothesis associated with edge  $e \in E$ .  
 Keep only relevant edges in  $E$ :
    - a. *check for verification criterion:*  
 $e \in E$  if and only if  $H_e$  is associated with verification criterion `ver` or `eff`, respectively.
    - b. *check for language constructs:*  
 $e \in E$  if and only if  $L_e \subseteq L_T$ .
  - 2: Compute  $\theta_{\text{ver/eff}}(o)$  according to [Equation 6.2](#) and return result.
- 

depending on the verification criterion (i.e., one for verifiability and one for verification effort). In [Algorithm 6.1](#), we summarize our computation of costs for individual options.

**Example 6.4.** Consider again [Figure 6.3](#). To compute the cost function  $\theta_{\text{eff}}$  focusing on verification effort, we first remove all edges associated with a hypothesis regarding verifiability (i.e., the directed edge  $(o_3, o_2)$ ). The score for each option  $o_1, o_2$ , and  $o_3$  then results in  $\theta_{\text{eff}}(o_1) = -\omega_1$ ,  $\theta_{\text{eff}}(o_2) = \omega_1 + \omega_3$ , and  $\theta_{\text{eff}}(o_3) = -\omega_3$ . Analogously, computing  $\theta_{\text{ver}}$  yields  $\theta_{\text{ver}}(o_1) = 0$ ,  $\theta_{\text{ver}}(o_2) = -\mathbb{1}_{L_T}\omega_2$ , and  $\theta_{\text{ver}}(o_3) = \mathbb{1}_{L_T}\omega_2$ . Regarding [Figure 6.3](#),  $\mathbb{1}_{L_T} = 1$  if and only if the analyzed program contains loops and 0 otherwise. The weights  $\omega_1, \omega_2$ , and  $\omega_3$  represent the effect size in  $[0.1, 0.5]$  for the particular hypothesis.

## Global Cost Optimization

After each configuration option  $o \in \mathcal{O}_S$  has been scored, GUIDO formulates an optimization problem and solves it by employing a linear programming solver [[Dantzig 1998](#)]. The main objective is to find the configuration candidate that maximizes the individual scores per configuration option based on our verification criterion. Essentially, our verification criterion is twofold; on the one hand, we try to increase verifiability, whereas on the other hand, we try to reduce verification effort. To span a continuum between these two, we introduce parameter  $\gamma$  to control our verification focus and require that  $\gamma \in [0, 1]$  with the following meaning: a setting of  $\gamma = 1.0$  focuses completely on verifiability, whereas a setting of  $\gamma = 0.0$  focuses completely on verification effort. A setting in between represents a mixed and weighted approach.

Solving the optimization problem to find an optimal configuration is realized by formulating a linear constrained-optimization problem, which can be solved with linear programming. Besides the objective that is subject to maximization, the configuration space has to be encoded (see the set of valid configurations  $\mathcal{C}_S$  described in [Section 6.2](#)). The encoding may look different for different verification systems, which we indicate by the placeholder function `encodedConfSpace(·)`. For instance, all considered control parameters in `KEY` have this property that their configuration options are mutually exclusive (i.e., for every  $p \in \mathcal{P}_S$  with  $p = \{o_1, \dots, o_n\}$  it holds that  $c(o_i) = 1$  if and only if  $\forall_{k \neq i} c(o_k) = 0$ ). The resulting formalization of the linear optimization problem with the highest rated configuration as a solution looks then as follows.

$$\begin{aligned}
c^* &= \arg \max_{c \in C} \sum_{\substack{\forall o \in \mathcal{O}_S \\ \wedge c(o)=1}} \gamma * \theta_{\text{ver}}(o) + (1 - \gamma) * \theta_{\text{eff}}(o) \\
\text{subject to} \quad & \text{encodedConfSpace}(c, p) \quad \forall c \in C, p \in \mathcal{P}_S \\
& c(o) \in \{0, 1\} \quad \forall c \in C, o \in \mathcal{O}_S
\end{aligned}$$

Considering again the typical scenario for a verification system where no constraints between configuration options of different parameter exist (i.e., exactly one configuration option of each parameter has to be selected), a definition of the encoding as a linear constraint looks as follows.

$$\text{encodedConfSpace}(c, p) := \forall c \in C, p \in \mathcal{P}_S : \left( \sum_{o \in p} c(o) \right) = 1 \quad (6.3)$$

If the currently optimal configuration is insufficient for proving the input task automatically, and a user-defined timeout  $\tau$  is not reached, GUIDO applies a *continuation mechanism*  $\mathcal{M}$  to compute another configuration for re-trying the verification attempt. In particular, the following three continuation mechanisms are currently supported by GUIDO.

**M<sub>penalty</sub> (Apply penalty).** With this mechanism, each active option of the current configuration is penalized by adding a constant  $k$  to its score. Afterwards, the constraint optimization problem is solved again. For instance, a score may be worsened by a penalty of  $k = -0.05$ . A *good*  $k$ , however, depends on the normalization of the scoring function. This penalty helps to overcome *overfitting*, as our dataset may not reflect necessary information to handle every program optimally. Options with a good score that would otherwise prevail will eventually be scored lower in case of repeated failure.

**M<sub>next</sub> (Next in rank).** With this mechanism, the configuration with the *next* highest score is chosen. This is particularly useful, if the parameter  $\gamma$  is currently in favor of the verification effort (i.e.,  $1 - \gamma > \gamma$ ), as the next best configuration will likely increase verifiability. Technically, a constraint is added to the optimization problem that removes the current configuration as a possible solution.

**M<sub>adjust</sub> (Adjust weights).** With this mechanism, weight  $\gamma$  is changed in favor of verifiability by a fixed constant  $k$  (i.e.,  $\gamma_{\text{NEW}} := \gamma_{\text{OLD}} + k$ ). For instance,  $k = 0.1$  would shift the focus in favor of provability by 10% each iteration. Similar to mechanism **M<sub>next</sub>**, the chances of a successful verification increase.

We postpone a discussion of the usefulness of each of these mechanisms to [Section 6.6](#), where we will evaluate whether one of these mechanisms is superior compared to the other two with respect to verifiability and verification effort.

### 6.3.3. Summary of Main Algorithm

So far, we gave an informal overview of GUIDO in [Section 6.2](#) and described GUIDO's technical core aspects in [Section 6.3](#). Whereas the offline phase has to be performed only once

---

Algorithm 6.2:  $\text{GUIDO}(\mathcal{H}_S, \mathcal{D}_S, T, \gamma, \alpha, \tau)$ , where  $\mathcal{H}_S$  is the set of statistical hypotheses,  $\mathcal{D}_S$  is the benchmark dataset,  $T$  is a verification task,  $\gamma \in [0, 1]$  is a real coefficient,  $\alpha \in [0, 1]$  is the significance level, and  $\tau$  represents the timeout.

---

- 1: **Offline (see Section 6.3.1):** If the set of hypotheses  $\mathcal{H}_S$  is not yet evaluated, test each statistical hypothesis in  $\mathcal{H}_S$  with significance  $\alpha$  on collected observations in  $\mathcal{D}_S$ . We then denote  $H^* \subseteq \mathcal{H}_S$  as the set of *accepted hypotheses*.
  - 2: Based on  $H^*$ , construct a *cost graph*  $G_p$  for each option group  $p \in \mathcal{P}_S$  that ranks the contained options with respect to the verification criterion.
- 
- 3: **Online (see Section 6.3.2):** Analyze the input task  $T$  for language constructs  $L_T \subseteq \mathcal{L}$ .
  - 4: Score each option  $o \in p$  by applying a scoring function with respect to  $L_T$

$$\theta_{\text{ver/eff}}(o) = \text{COMPUTESCORE}(o, G_p, L_T).$$

- 5: While verification fails with configuration  $c$  and timeout  $\tau$  is not yet reached:
    - a. Solve constrained linear optimization problem for the optimal configuration while respecting the configuration space
 
$$c^* = \text{SOLVECOP}(\theta, \gamma)$$
    - b. In case of failure, use a continuation mechanism  $\mathcal{M}$  to compute the next configuration.
  - 6: Use the current optimal configuration for verification. In case of failure, report applied configurations for inspection.
- 

for a particular release of a verification system, the online phase is applied individually to every input verification task based on the constructed cost graphs. In [Algorithm 6.2](#), we summarize the main algorithm of GUIDO.

Only few parameters need to be tuned for GUIDO. First,  $\gamma \in [0, 1]$  is able to shift the focus of the configuration search from only considering verification effort ( $\gamma = 0.0$ ) to only considering verifiability ( $\gamma = 1.0$ ). We assume that a linear trade-off for the program verifier KEY exists between the two criteria. In [Section 6.6](#), we will evaluate (1) whether this assumption holds and (2) if it holds, which value of  $\gamma$  represents a reasonable trade-off between both criteria. Second, significance level  $\alpha$  must be adjusted in the offline phase. Although GUIDO applies an additional correction to the significance level, a user-defined value around 0.05 is expected to produce reasonable results. Third, some of the continuation mechanisms depend on some constant  $k$ , which must be defined by end-users and potentially need to be adapted to the benchmark dataset and verification task for optimal results. GUIDO comes with default settings for the presented continuation mechanisms, which provide adequate results based on our experience and the verification tasks we considered throughout this chapter. Finally,  $\tau$  represents the maximum time that GUIDO spends on the configuration search, and may therefore vary greatly for different verification systems and even the input verification task. This includes the verification effort itself and the configuration search

through solving the linear optimization problem. We assume that end-users have at least a rough intuition on the time needed to verify specific verification tasks.

## 6.4. Open-Source Implementation

We implemented GUIDO in JAVA to quantitatively evaluate our technique. To model the configuration space of KEY-2.7.0, we exploit the configuration utilities of the open-source tool FEATUREIDE 3.5.3 [Meinicke et al. 2016; Meinicke et al. 2017], an extensible framework for the development of software product lines [Pohl et al. 2005; Czarnecki et al. 2000]. FEATUREIDE allows us to model the configuration space as a feature diagram [K. C. Kang et al. 1990], a graphical and easy-to-understand modeling language, and already supports t-wise sampling.

The key component for establishing the benchmark dataset in the offline phase is the automated `VerificationServer`. As verification in itself is very costly and the number of verification attempts can be very large, the `VerificationServer` is based on a *client-server* architecture. The server manages a queue of verification attempts and distributes them to a number of started clients. This allows to compute results in parallel. Moreover, each verification attempt needs to be performed independently to not falsify the results, since a verification system could store information about previous verification tasks. To mitigate a falsification of the result, each client (1) performs one verification attempt in a virtual machine, (2) starts a new independent client afterwards, and (3) eventually terminates itself.

Besides a *client-server* architecture for automatically establishing the benchmark dataset in the offline phase, GUIDO's implementation is accompanied with language support for formulating the hypotheses in the form of an ECLIPSE plug-in. In Listing 6.2, we exemplify an excerpt of such language support for organizing hypotheses. This example defines a hypothesis for parameter `proofSplitting`,

---

```

1 Import : "others.hypothesis"
2 Dataset : "data.txt"
3 System : KeY-2.7.0
4
5
6 Hypothesis H3 on Parameter proofSplitting {
7   Description : "proofSplitting.delayed might be slower!";
8   Definition : {proofSplitting.off: true} is more efficient than {
9     proofSplitting.delayed: true};
9   Constructs : [LC.NO_IF];
10 }
11
12 // ... other hypotheses ...

```

---

Listing 6.2: Excerpt of domain-specific language for organizing hypotheses.

which states that `off` decrease verification effort compared to `delayed` in the absence of branching.

With respect to the benchmark dataset, hypotheses and all experimental data are translated to R<sup>5</sup> and automatically evaluated to determine the set of accepted hypotheses. For solving the lin-

---

<sup>5</sup><https://www.r-project.org/>

Statistic	Value	Statistic	Value
Configuration options	56	Fixed configuration options	15
Parameters	30	Significance level $\alpha$	0.05
Defined hypotheses	72	Accepted Hypotheses	12–16
Total configurations	663,552	Verification tasks	94
Sample size ( $t = 3$ )	2,235	Total verification attempts	210,090

Table 6.3.: Statistics on the Application to KEY

ear optimization problem, which is formulated and solved in the online phase, we employ *SCP-Solver*, a JAVA-based solver for linear programs.<sup>6</sup>

The online phase is implemented by three main components, namely the *CodeAnalyzer*, *ScoreGenerator*, and *ConfigurationGenerator* (see Figure 6.2 and detailed description in Section 6.3). GUIDO receives the accepted hypotheses, optional parameters (i.e., weight  $\gamma$  and constants for timeout and adjustment), and verification tasks as input, and computes promising configurations for each verification task or provides feedback to the user for manual inspection. In component *ConfigurationGenerator*, the linear optimization problem is formalized and solved. To transfer GUIDO to other verification tools, only some adaptations are needed.

While we demonstrated GUIDO’s application primarily in the context of deductive verification and the program verifier KEY (see Section 6.5), the current implementation also integrates the configurable model checker CPACHECKER-1.8 [Beyer et al. 2011]. We provide an open-access repository of GUIDO including hypotheses and benchmark data sets for KEY-2.7.0 and CPACHECKER-1.8.<sup>7</sup>

## 6.5. Illustrative Application on Deductive Program verification with KEY

We applied GUIDO to the deductive program verifier KEY-2.7.0 [Ahrendt et al. 2016]. KEY-2.7.0 comprises a total of 30 parameters, where each parameter is associated with *two* to *four* configuration options. Checkboxes constitute binary parameters consisting of two configuration options, namely *true* and *false*. In KEY, there exist no further constraints between configuration options from different parameters, which is why the encoded configuration space is equal to Equation 6.3 (see Section 6.3.2). We summarize the most important statistics of applying GUIDO to KEY in Table 6.3, which we explain in more detail in the following.

Typically, numerous configuration options not only tune the verification algorithm itself, but also may change *what* is verified (e.g., absence of overflows, when integers are treated with the semantics of the programming language). Consequently, a number of options can be fixed in the beginning to already reduce the configuration space. As we only apply KEY to JAVA programs, we fixed options that would either prohibit a successful verification of such programs or would falsify the result. For instance, we always applied the *Java semantics* for integer values instead of treating integers as mathematical objects with infinite domains. In total, we fixed 15 configuration options, such that

<sup>6</sup><http://scpsolver.org/>

<sup>7</sup><https://github.com/AlexanderKnueppel/Guido>

663,552 different configurations remain. To further reduce the configuration space, we used *three-wise* sampling (i.e.,  $t = 3$ ) employing the ICPL sampling algorithm [Johansen et al. 2012](#), which is state-of-the-art for large configuration spaces [\[Varshosaz et al. 2018\]](#). As a result, 2,235 configurations remain. Consequently, we performed a total of 210,090 verification attempts (i.e., each configuration sample is applied to each verification task) to compute the benchmark dataset.

For deductive verification, our metric for verification effort is twofold: either (1) execution time of the verifier, or (2) the size (i.e., number of steps) of generated proofs. Finding configurations that lead to a reduced proof size may help in two ways. First, they are resource-beneficial in case of distribution, such as with proof-carrying code [\[Necula 1997\]](#). Second, as proofs can be replayed in re-verification attempts (e.g., when applied in continuous integration), smaller proofs are replayed faster and, thus, accumulate to less verification time [\[Beckert et al. 2004; Hähnle et al. 2013\]](#).

To capture our domain knowledge, we have formulated 72 hypotheses about verifiability and verification effort, and investigated each hypothesis as one independent experiment. As there is a high chance that execution time and number of proof steps correlate under ideal conditions, we evaluated the same 72 hypotheses regardless of the applied metric for effort. In each of our hypotheses, we only test the effect of two mutually exclusive options, as we are not aware of any interactions between options in KEY. As we perform 72 different experiments on the same benchmark dataset, we apply the *Bonferroni correction* (see hypothesis testing in [Section 6.3.1](#)). We set our initial significance level to the commonly practiced 5%. However, it is noteworthy that for the context of formal verification, a higher significance level may be more promising, as more hypotheses can be accepted and therefore more information on the influence of some options is available. Applying the correction yields  $\alpha = 0.05/72 \approx 0.0007$ . All null hypotheses with a p-value lower than this significance level  $\alpha$  were rejected, meaning the alternative hypotheses were accepted. As further elaborated in our evaluation (see [Section 6.6](#)), we accepted 12–16 hypotheses, depending on the benchmark verification tasks and the applied metric for verification effort we considered during the hypotheses testing.

## 6.6. Evaluation

With GUIDO, we aim at automatically generating configurations for configurable program verifiers that lead to an acceptable trade-off between *verification effort* and *verifiability*. Meeting this goal depends on GUIDO’s practicality and capabilities as presented in [Section 6.3](#) to improve the automatic verification process. Based on our tool support presented in the previous section (see [Section 6.4](#)), we empirically evaluate GUIDO by addressing the following five research questions.

**RQ-1:** *What impact has parameter  $\gamma$  on verifiability and verification effort?*

**RQ-2:** *Which continuation mechanism performs best after a failed verification attempt?*

**RQ-3:** *How does GUIDO compare to KEY’s default configuration or a trial-and-error strategy?*

**RQ-4:** *To what extent can GUIDO reduce the proof size of verification subjects systematically?*

**RQ-5:** *For model checking, how does GUIDO compare to the default configuration of CPACHECKER-1.8?*



Project	Classes	Methods	Tasks	Where?
OpenJDK	3	15	23	[Knüppel et al. 2018a]
KEY Examples	12	24	25	[Ahrendt et al. 2016]
BankAccount-FH-JML	3	11	11	[Thüm et al. 2019]
OpenJMLDemo <sup>8</sup>	10	35	35	[Cok 2011]

Table 6.4.: Corpus of JML projects and their characteristics used for evaluating GUIDO on KEY-2.7.0.

The first four research questions focus primarily on deductive verification employing the program verifier KEY-2.7.0 as described in Section 6.5. The fifth question addresses a different application scenario, namely model checking, and is evaluated on CPACHECKER-1.8. For all measurements, we used an Intel Core i7-7700K @ 4.20 GHz processor with 32 GB of RAM and Windows 10 on 64bit.

We elaborate on our methodology and introduce the experimental subjects in Section 6.6.1. Next, in Section 6.6.2, we present and discuss our results. Finally, in Section 6.6.3, we discuss potential threats undermining the validity of our study.

### 6.6.1. Methodology and Evaluated Projects

As mentioned before, **RQ-1–RQ-4** are directly targeted towards deductive verification in general and KEY-2.7.0 in particular. We evaluate GUIDO on a corpus of 94 verification tasks (i.e., specification cases) provided in JML from widely known projects. While 94 experimental subjects do not constitute a large number of programs, specifying programs that can be verified automatically poses an immense challenge in the first place [Baumann et al. 2012; Rozier 2016; Knüppel et al. 2018a]. Therefore, this initial evaluation of GUIDO is conducted on only a small number of experimental subjects, which, nonetheless, will allow us to address the research questions raised above. Table 6.4 illustrates the most important characteristics of these experimental subjects, including the projects themselves and number of classes, methods, and resulting verification tasks. The benchmark dataset is generated on the aforementioned 94 specification cases with a sampled set of 2,235 configurations, yielding 210,090 data entries in total (cf. Section 6.5).

**Setup.** For each experiment, we performed a ten-fold cross validation. That is, we partitioned the 94 verification tasks into 90% benchmark verification tasks (i.e., 84 or 85 tasks) and 10% test input (i.e., nine or ten tasks) over the course of ten rounds, such that all benchmark verification tasks served as test input once. This means that we had to re-evaluate the hypotheses for each set of training verification tasks. For **RQ-1–RQ-3**, we accepted the same 16 hypotheses in eight of the ten rounds with a slight variance in the p-values. In two rounds, only 13 and 14 hypotheses were accepted, respectively. We use a timeout of five minutes for each individual verification attempt to keep the experiments reasonable in time. We found that five minutes are enough to successfully verify the majority of our verification tasks. To mitigate computation bias influencing execution time, we conducted all respective experiments 10 times and picked the median. For **RQ-4**, we use the proof size as the performance metric for the verification effort instead of the execution time. Changing

<sup>8</sup><https://github.com/OpenJML/OpenJMLDemo>



the metric has a direct influence on the hypotheses about verification effort. Consequently, we only accepted the same 15 hypotheses in nine rounds and 12 hypotheses in one round.

To evaluate generalizability of our results, we designed an experiment (**RQ-5**), in which we applied GUIDO to CPACHECKER-1.8 [Beyer et al. 2011], a configurable model checker developed in JAVA for verifying C programs. In our configuration space, we encoded 18 parameters with a total of 41 configuration options. For this study, we decided to only compare against the default analysis of CPACHECKER (e.g., assertion checking and termination), and to use the *total time* spent in milliseconds as a representative for the verification effort. We chose a total of 30 benchmarks with varying characteristics (i.e., loops, arrays, small, and large), which we randomly divided into a single training set (23 benchmarks) and a test set (seven benchmarks).<sup>9</sup> Our established data set comprises approximately 86,440 data entries (i.e., 3,758 sampled configurations applied to each of the 23 benchmarks). We formulated 14 hypotheses based on the online documentation and scientific publications, of which we could accept 11. More information on the hypotheses can be found in [Appendix B](#).

**Baseline.** For research question **RQ-1**, we set weight  $\gamma$  to eleven uniformly distributed values between 0.0 and 1.0 (i.e., with step size 0.1) and measure the effect on verification effort and verifiability. If a higher value of  $\gamma$  leads to significantly more successful proof attempts, but a lower value leads to significantly less execution time for the proven cases, we may conclude that a trade-off between verifiability and effort exists. Answering this question is necessary to justify our linearization between both verifiability and verification effort, and our continuation in case of failure. For research question **RQ-2**, we compare GUIDO to the *default configuration* of KEY and a *trial-and-error strategy* (i.e., randomly picking a configuration from the configuration space) to compute the next configuration in case the default one is insufficient. This may mimic the behavior of an inexperienced user, who starts with the default settings and changes them in a trial-and-error fashion after unsuccessful verification attempts. We assume that the default configuration is already a good configuration in terms of provability for easier verification problems, but may not be the fastest configuration and also not sufficient for more complex problems. For **RQ-3**, we evaluate whether one continuation mechanism with respect to KEY and our experimental subjects is superior (i.e., more effective or more efficient) to another mechanism. For **RQ-4**, we evaluate GUIDO’s potential to reduce the size of a proof measured in proof steps with respect to KEY’s own proof format and compare the result to the aforementioned try-and-error strategy starting from the default configuration. For **RQ-5**, we compare GUIDO with the default configuration of CPACHECKER on seven benchmarks that vary in language constructs and size. Although we expect that the default configuration is an adequate candidate for these benchmarks, we evaluate whether GUIDO may significantly reduce the verification effort for a subset of them.

### 6.6.2. Results and Insights

In the following, we discuss all five research questions raised above.

<sup>9</sup><https://github.com/sosy-lab/sv-benchmarks>

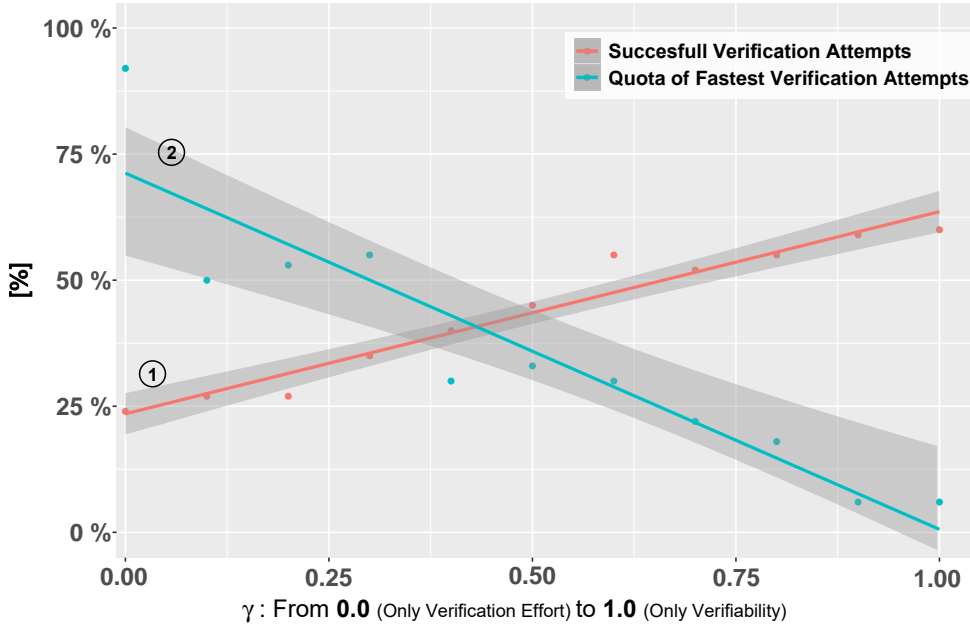


Figure 6.4.: Percentage of verified tasks and compared efficiency for different values of  $\gamma \in [0, 1]$ .

### RQ-1: Influence of Parameter $\gamma$ on Verifiability and Verification Effort

With this research question, we use GUIDO to investigate whether a trade-off between verifiability and verification effort is observable in our experimental subjects with respect to different configurations. In Figure 6.4, we visualize how changing the focus between verifiability and verification effort (i.e., adjusting weight  $\gamma$ ) impacts the verification result. We verified each of the 94 subjects a total of eleven times using GUIDO, starting with  $\gamma = 0.0$  (complete focus on verification effort). Each iteration, we increased  $\gamma$  by a step size of 0.1 and repeated the experiments. As a result, we obtain two trend lines that we explain in the following.

First, the trend line colored in red ① illustrates the total percentage of successfully verified tasks with each value of  $\gamma \in [0, 1]$  and step size 0.1. For instance,  $\gamma = 0.5$  results in roughly 46.8% successfully verified tasks (i.e., 44 verification tasks). A value of  $\gamma = 0.0$  leads to the minimum number of verified tasks in the given time span (24%), whereas a value of  $\gamma = 1.0$  leads to the maximum number of verified tasks (67%). Second, the trend line colored in cyan ② illustrates the percentage of the tasks that (1) were successfully verified and (2) needed the least amount of execution time for any  $\gamma$ . That is, the percentage indicates to what extent a particular  $\gamma$  led to a configuration that was able to complete a proof *and* computed a proof the fastest compared to all other values for  $\gamma$ . For instance, setting  $\gamma = 0.5$  means that 33% of the verification tasks were successfully verified and the configurations computed by GUIDO for these tasks and  $\gamma = 0.5$  also minimized verification effort.

**Discussion.** In summary, complete focus on verifiability (i.e.,  $\gamma = 1.0$ ) lets GUIDO produce configurations that allow to verify a total 63 out of 94 tasks automatically (i.e., 67%), whereas with complete focus on verification effort (i.e.,  $\gamma = 0.0$ ), only 23 tasks (i.e., 24%) were verified successfully. However, applying  $\gamma = 0.0$ , GUIDO was able to minimize the execution time for 21 tasks (i.e., 33% of

Mechanism	No. Successful Tasks	Success Rate [%]	∅ Time [ms]
$M_{\text{penalty}}$	60	63.83%	4,726
$M_{\text{adjust}}$	63	67.02%	3,094
$M_{\text{next}}$	54	57.43%	5,122

Table 6.5.: Statistics on successfully verified tasks for each continuation mechanism.

the 63 successful verification attempts). This experiment raises our confidence that a linear trade-off between verifiability and verification effort exists. That is, optimizing both verifiability and verification effort at the same time is for many verification tasks impossible. These insights allow us to tune the configuration search further and to minimize the number of iterations for GUIDO. Although minimizing verification effort is important in later stages of the development process, the primary goal is typically to automatically and successfully verify programs in the first place. Regarding our experimental subjects, a good starting point is therefore to uniformly weight the influence of verifiability and verification effort for the configuration search (i.e., setting  $\gamma = 0.5$ ), as this increases the success rate compared to a complete focus on verification effort by 21%, but also generates in 33% of successful attempts a proof while minimizing verification effort.

## RQ-2: Comparison of Continuation Mechanisms

With this research question, we evaluate the influence of the three proposed continuation mechanisms (see Section 6.3.2). That is, we compare the three different mechanisms, namely  $M_{\text{penalty}}$ ,  $M_{\text{adjust}}$ , and  $M_{\text{next}}$ , of GUIDO in terms of verifiability and verification effort. We set  $\gamma = 0.5$ , as we evaluated before that this value provides a good compromise with respect to effectiveness and performance. For  $M_{\text{adjust}}$  and  $M_{\text{penalty}}$ , we use  $k = 0.1$  and  $k = -0.05$ , respectively.

In Table 6.5, we illustrate each mechanism with the total number of successful verification attempts after ten rounds and the average computation time per successful verification attempt.  $M_{\text{adjust}}$  is able to verify the most tasks with roughly 67% in the given time frame of 1 minute.  $M_{\text{penalty}}$  and  $M_{\text{next}}$  close slightly less proofs (i.e., 64% and 57%, respectively). Moreover,  $M_{\text{adjust}}$  needed the least amount of time on average to close a proof. This is due to the fact that the adjustment oftentimes needs less iterations compared to the other two strategies, as the focus on verifiability increases quickly.

In Figure 6.5, we illustrate the produced proof sizes for all verification attempts in a box plot. As can be seen,  $M_{\text{penalty}}$  and  $M_{\text{next}}$  produce similar results, whereas  $M_{\text{adjust}}$  produces slightly larger proofs. This is due to the fact that  $M_{\text{adjust}}$  (1) verified more tasks in total and (2) could successfully close more complex proofs.

**Discussion.** In Figure 6.4, we show that GUIDO’s first attempt with  $\gamma = 0.5$  leads to 44 successfully verified tasks (46.8%). Therefore,  $M_{\text{next}}$  is able to verify 10 more tasks,  $M_{\text{penalty}}$  is able to verify 16 more tasks, and  $M_{\text{adjust}}$  is able to verify 19 more tasks. In particular, this experiment shows that high values of  $\gamma$  lead to configurations that maximize verifiability, which further increases confidence in our linearization. We conducted a manually performed statistical hypothesis test to investigate whether a mechanism is superior to the other two mechanisms. However, based

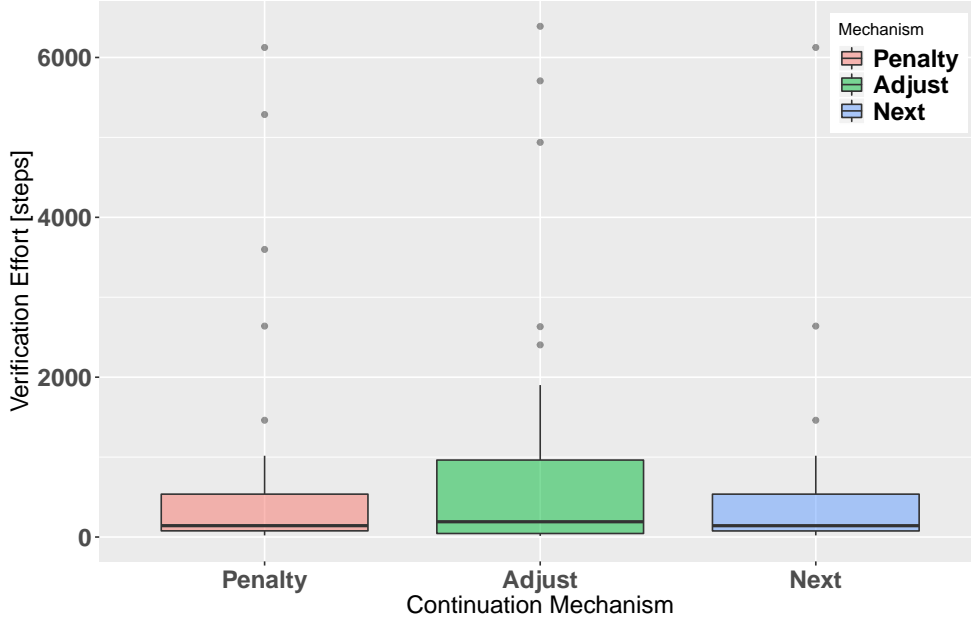


Figure 6.5.: Percentage of closed proofs for the three different continuation strategies.

on our measurements, we could not accept the hypothesis that there exists one mechanism that is significantly better ( $\alpha = 0.05$ ) than the other two mechanisms. Nonetheless, we conclude for **RQ-3** and our experimental subjects that  $\mathbf{M}_{\text{adjust}}$  should be used here as the default mechanism, as it represents a slightly better trade-off between verifiability and verification effort for  $\gamma = 0.5$  compared to the other two strategies. We acknowledge that other datasets may greatly benefit from different (and even more sophisticated) strategies, which should be investigated further in future evaluations. Moreover, we fixed parameter  $k$  for  $\mathbf{M}_{\text{penalty}}$  and  $\mathbf{M}_{\text{adjust}}$  to a single value. Other values may lead to other insights, which we also may investigate in the future.

### RQ-3: Effectiveness and Efficiency of GUIDO

With this research question, we investigate whether GUIDO can outperform the default settings of KEY-2.7.0. In [Figure 6.6](#), we compare the performance of GUIDO by using a *trial-and-error* strategy as baseline. Essentially, the trial-and-error is a combination of KEY's own default configuration<sup>10</sup> and a randomized continuation mechanism. Initially, the trial-and-error strategy starts with the default configuration. That is, if the very first verification attempt is successful, the result was produced using KEY's default configuration. In case of failure, the trial-and-error strategy continues with flipping a random configuration option to generate a *new* configuration from the configuration space that is still close to the configuration before.

Configurations in GUIDO and the trial-and-error strategy are generated until either (1) a verification task is verified successfully, (2) a timeout of five minutes was reached, or (3) a predefined number of verification attempts is exceeded (i.e., maximum number of tries). As concluded in **RQ-1**, we

<sup>10</sup>The user interface of KEY offers some configuration profiles to choose from. We refer to the *default configuration* as the profile with name **default**, which is also the initial configuration when KEY is started the first time. It is important to note that default configurations may vary between versions of KEY.

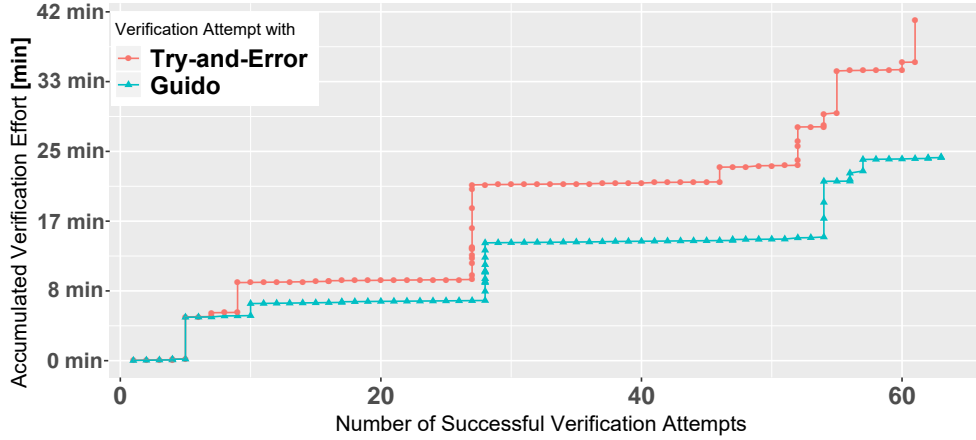


Figure 6.6.: Performed verification attempts and accumulated effort for GUIDO and a trial-and-error strategy (adopted from Knüppel et al. [2021b]).

set  $\gamma = 0.5$  to weight verification effort and verifiability uniformly. Moreover, we apply  $\mathbf{M}_{\text{adjust}}$  as continuation mechanism for GUIDO, which we concluded in **RQ-2** to produce sufficient results for our experimental subjects. In case of failure, we increase  $\gamma$  by 0.1, which increases the probability of successful verification. This also means that GUIDO tries a maximum number of six configurations for any verification task. Consequently, to enable a fairer comparison, we limited the number of tried configurations for the trial-and-error strategy also to six.

Each point in Figure 6.6 represents one verification attempt. On the horizontal axis, we depict the number of successful verification attempts for both strategies. On the vertical axis, we accumulate the spent verification effort measured in execution time. Both graphs are shaped like *step functions*. Vertically-arranged points in the step function indicate that the respective strategy was insufficient to verify the particular verification task. GUIDO successfully verified 63 verification tasks, whereas the trial-and-error strategy verified only 61 verified tasks. Considering all 94 verification tasks, GUIDO spent approximately 26 minutes on average to try verifying all of them, whereas the trial-and-error strategy spent approximately 40 minutes, which is an increase of 63% in verification effort.

**Discussion.** Based on our verification subjects, this experiment illustrates that GUIDO can be more effective for the deductive program verifier KEY-2.7.0 than our baseline trial-and-error strategy (i.e., GUIDO verifies an additional two tasks), while also being more efficient (i.e., GUIDO reduces verification effort by 35%). However, it also affirms that the default configuration is already very effective in verifying the majority of verification tasks automatically, which is visualized by all single dots in Figure 6.6 that are not vertically arranged with other dots. An advantage of GUIDO is that costly configuration options that do not increase verifiability are oftentimes ignored, whereas the trial-and-error strategy may pick such configuration options by chance. That is, even if a verification task is not verifiable, GUIDO tends to generate configurations that *fail faster* compared to the trial-and-error strategy. This is particularly important, as failed verification attempts oftentimes provide little insights on the reason, which could be non-conformance of specification and implementation, but also due to insufficient parameterization. With GUIDO, the reason of insufficient configurations can be minimized.

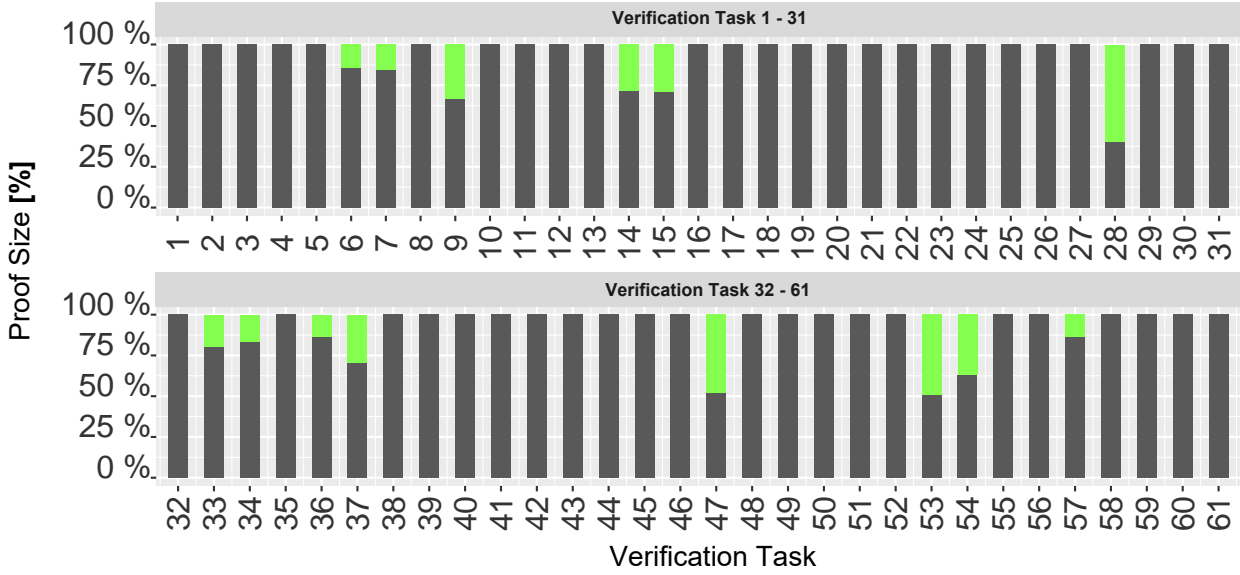


Figure 6.7.: Reduction in proof size for 14 verification tasks when applying GUIDO instead of the try-and-error strategy. Percentage of saved proof steps is highlighted .

Additionally, the accumulated verification effort for GUIDO also included solving the underlying optimization problem. Consequently, this experiment shows that for KEY’s configuration space and the number of accepted hypotheses, the overhead of solving the linear optimization problem is of little importance compared to the improvements in verification effort. To answer **RQ-3** with respect to our experimental subjects, GUIDO outperforms both the default strategy and the random continuation.

#### RQ-4: Effectiveness and Efficiency of GUIDO

With **RQ-3**, we evaluated how effective GUIDO is in finding configurations with respect to time as a measurement for the verification effort. With **RQ-4**, we compare to what extent GUIDO is able to reduce the *size of a proof* (i.e., the number of proof steps), which we use in this experiment as the measurement for the verification effort. To this end, we apply GUIDO on all 61 tasks of **RQ-3** that were verified by both GUIDO and the trial-and-error strategy, and adjust weight  $\gamma$  in favor of verification effort until a task is not verifiable anymore. In Figure 6.7, we illustrate our results for all 61 verified tasks. Compared to the trial-and-error strategy, GUIDO was able to reduce the size of 14 proofs, saving 13% to 59% proof steps (31% on average). For all other tasks, both strategies found the same minimal proof.

**Discussion.** When proofs are used in proof-carrying code [Necula 1997], or proof replay [Beckert et al. 2004; Hähnle et al. 2013] is applied frequently (as relevant for continuous integration), the size of a proof is an important metric. In this experiment, GUIDO outperformed the trial-and-error strategy by reducing 23% of generated proofs significantly (i.e., up to 59%) and not a single proof was larger. This result showcases GUIDO’s ability to reduce verification effort in a different dimension (i.e., proof steps in contrast to execution time). To conclude for **RQ-4**, this experiment raises

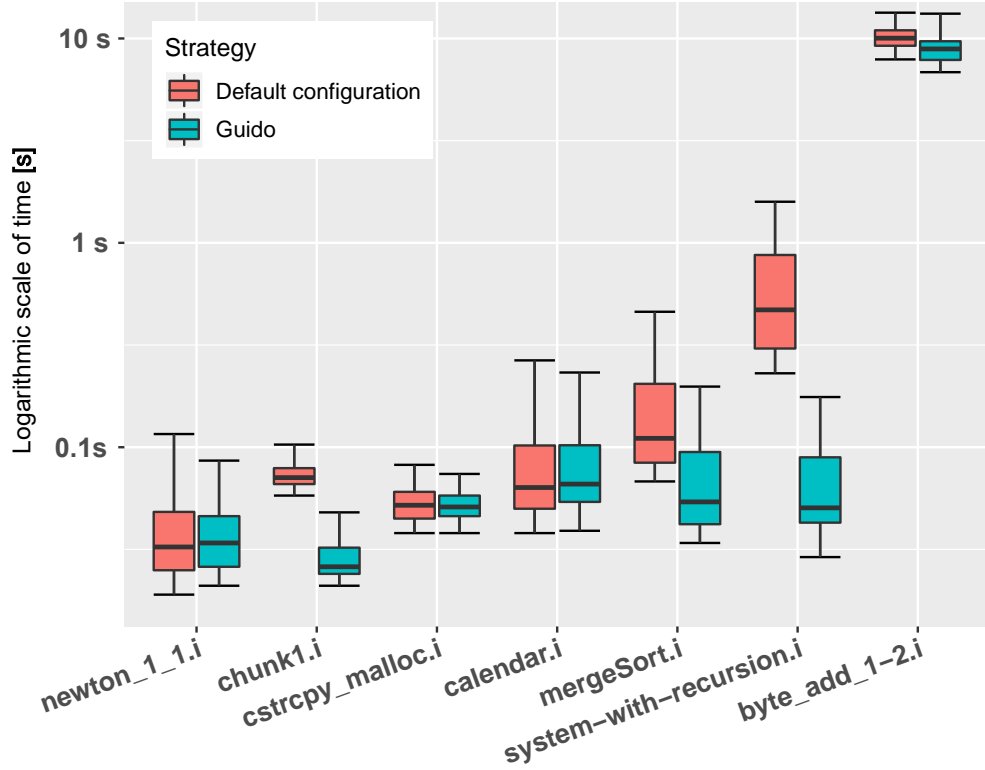


Figure 6.8.: Compared efficiency for GUIDO and the default configuration of CPACHECKER-1.8 on seven benchmarks.

our confidence that we are able to purposefully reduce the verification effort by incorporating domain knowledge in the process. However, as we are limited to our verification subjects, it is necessary to evaluate GUIDO on more complex problems in the future.

## RQ-5: Comparison with CPACHECKER

So far, we primarily focused on deductive verification within this line of research. However, we hypothesized that GUIDO’s core concepts can probably be generalized to other types of formal verification with configurable program verifiers, such as model checking. With this research question, we aim at applying GUIDO to the model checker CPACHECKER-1.8 to get a first impression on GUIDO’s performance in a different application scenario. In Figure 6.8, we compare GUIDO to the default configuration of CPACHECKER with respect to the verification effort. The effort is depicted on the vertical axis and measured in seconds representing the total time spent by CPACHECKER analyzing and solving the verification task. To mitigate computational bias, each box plot comprises a total of 100 runs. For this experiment,  $\gamma$  had not much of an influence, as (1) only two accepted hypotheses focused on verifiability and (2) all seven benchmarks were verifiable with the vast majority of generated configurations. We therefore fixed  $\gamma$  again to a value of 0.5 and did not apply any continuation mechanism. As a result, GUIDO reduced the verification effort for four of the seven programs significantly: `chunk1.i` (p-value < 2.2e-16), `mergeSort.i` (p-value = 4.403e-15), `system-with-recursion.i` (p-value < 2.2e-16), and `byte_add_1-2.i` (p-value = 7.367e-15).



**Discussion.** For **RQ-5**, this experiment raises our confidence that we are able to apply GUIDO to more than one domain of configurable formal verification tools. Although only four of the seven evaluated verification tasks benefited significantly from applying GUIDO, even better results are expected for a more sophisticated set of hypotheses and a larger dataset, as only 28 benchmark verification tasks were used to compute the cost model.

We only focused on performance, as the analyzed benchmarks were all simple enough to be verified with either the default configuration or the configuration generated by GUIDO. However, time is a crucial aspect of model checking, which often suffers from the state space explosion problem (see [Section 2.1.2](#)). We therefore expect that many unsuccessful verification attempts of more complex benchmarks in the context of model checking stem from timeouts or other limited resources. With GUIDO, configurations that minimize verification effort can be identified, which will likely increase effectiveness as well. Moreover, GUIDO is agnostic to the concrete measurement used for verification effort. In the context of model checking, memory consumption is another interesting performance metric to optimize.

### 6.6.3. Threats to Validity

Results of our evaluation are confronted with several threats to validity that we discuss in the following.

**Internal Validity.** In our evaluation, we used a fixed timeout of 5 minutes for GUIDO to find a suitable configuration. The verification time needed depends on many factors, such as computing power and parallel processes. Thus, verifying the same verification task may sometimes fail due to the timeout. We mitigated this problem by reducing the number of parallel processes to a minimum and performed the experiments 20 times to omit outliers. For **RQ-4**, we considered the proof steps of a proof as a metric for the effort instead of the verification time. Proof steps may only be considered for verification systems build on proof systems and not a useful metric for every end-user. However, for KEY in particular, the number of proofs steps is a typical metric for the verification effort in the literature [[Knüppel et al. 2018c](#); [Bubel et al. 2014](#); [Beckert et al. 2004](#)]. For other verification systems, we abstract from a concrete measurement in our formalization.

Our implementation of GUIDO and our evaluation scripts may have bugs. We performed tests and extensive code reviews to rule out any serious bug. Moreover, we manually and repeatedly validated samples of the computed scores and resulting configurations to assure that the implementation complies to our theoretical framework.

For KEY, we fixed a total number of 15 configuration options, which may diminish the influence they may have on verifiability and verification effort. However, a certain degree of domain knowledge is required by end-users to successfully employ a verification system in the first place (e.g., selecting options that influence *what* is verified). To address this threat, we (1) only fixed options for which a different setting would not be reasonable for JAVA program verification, as KEY can also be used as a first-order theorem prover, and (2) fixed configuration options that are part of the default configuration.

To sample our configuration space, we applied *t-wise* sampling using the ICPL algorithm proposed by Johansen et al. [[2012](#)]. Although there exist other sampling strategies that could

have been applied, we applied ICPL, as it is one of the most evaluated algorithms in the literature [Varshosaz et al. 2018].

Our designed try-and-error strategy used as a baseline in **RQ-2** and **RQ-3** simply applies KEY’s default configuration and – in case of failure – continues with a random configuration picked from the configuration space. It can thus be argued that the comparison with GUIDO is not fair, as the random selection process does not establish a list of ranked configurations. However, to the best of our knowledge, there does not exist any state-of-the-art approach that can be used as a baseline. Hence, our initial goal for this new line of research was to demonstrate that GUIDO performs better than the both easiest strategies, namely (1) applying the default configuration and (2) randomly choosing configurations. We explicitly do not rule out that there might emerge more sophisticated approaches in the future, which can be used for comparison.

**External Validity.** The selection of the verification subjects may affect the external validity, as they may not be representative for all tasks. Therefore, we analyzed a total of 94 verification tasks for KEY, which is more than the vast majority of related work on deductive verification considers due to the specification effort. We additionally mitigated this threat by using well-known projects from the literature that represent a wide variety of typical programming constructs and can mostly be verified automatically, which was a necessity for our evaluation. We also increased the confidence in our external validity by performing a ten-fold cross validation to alleviate any introduced computation bias and received surprisingly similar results. Our experiments showed that only 84 benchmark verification tasks were sufficient for improving the effectiveness of the configuration search significantly.

Moreover, we focused our evaluation mainly on KEY and deductive verification. Other systems or verification disciplines may behave differently. We chose KEY, as (1) it is the system and context we have most experience with, which simplified the task of formulating good hypotheses, and (2) it is a state-of-the-art program verifier with a large and active community, which may benefit from our insights. We additionally concentrated on verification effort represented by the proof’s size, which is best demonstrated on deductive verification systems with adequate proof management (i.e., such as KEY). Furthermore, this is – to the best of our knowledge – the first work that proposes an approach to systematically automate the configuration process in the context of deductive verification. To show application to other contexts, we applied GUIDO on the model checker CPACHECKER.

## 6.7. Related Work

In the following, we discuss related work close to the topic of this chapter, namely parameter tuning in general, configuration sampling, and configurable verification systems.

### Performance Prediction and Parameter Tuning

GUIDO reimagines the challenge of automatic configuration for software systems in the context of formal verification. At the same time, general performance prediction of configurable software is a highly researched area [Siegmund et al. 2015]. Siegmund et al. [2012] introduced SPLCONQUEROR, a state-of-the-art framework using regression-based machine learning to predict the performance of various configurations. In particular, SPLCONQUEROR is used beyond *software product*

lines by a many researchers to get insights on the influence of non-functional properties in configurable software [Grebhahn et al. 2014; Guo et al. 2013; Kienzle et al. 2016; Olacchia et al. 2012]. Other frameworks for performance prediction include CART/DECART [Guo et al. 2018], FOURIERLEARNING [Y. Zhang et al. 2015], and DEEPPERF [Ha et al. 2019]. Although our approach comes with the cost of manually formulating explicit domain knowledge, the aforementioned general-purpose prediction frameworks fall short in at least three categories. First, as they learn knowledge about the influence of parameters on their own, they presumably need more data than GUIDO. Second, they give no insights on why a particular configuration option is more relevant, as typical for fully-automated machine learning algorithms. In contrast, GUIDO allows to test the plausibility of the user-defined domain knowledge with statistical hypothesis testing. Finally, if verification fails, there is no continuation mechanism applied.

Another widely researched area is optimizing the selection of parameters of configurable software. Several statistical models [Yigitbasi et al. 2013; Zheng et al. 2007; Nair et al. 2018] and search-based optimization strategies [Henard et al. 2015; Wu et al. 2015] have been considered for the parameter tuning of configurable software. Moreover, optimization algorithms (e.g., Bayesian optimization [Jamshidi et al. 2016], hill climbing [Xi et al. 2004], or multi-objective optimization [Filiari et al. 2015]) to find optimal configurations using only a small number of samples have been studied. Although GUIDO yields already promising results, we acknowledge that a combination with other statistical models may lead to even better performance in the configuration search and should be investigated in future work.

## Configuration Sampling

To sample the configuration space, we applied *three-wise* sampling using ICPL. A better sample could reduce effort in the offline phase, if the sample is smaller. A more representative sample could more precisely identify the relevant hypotheses improving results of the online phase. There exist other sampling strategies that can be applied, such as INCLING [Al-Hajjaji et al. 2016], CHVATAL [Johansen et al. 2011], or CASA [Garvin et al. 2011]. Kaltenecker et al. [2019] proposed *distance-based sampling* as a useful sampling strategy with lower error rates for large sample sizes. Most of these sampling techniques use only the model of the configuration space as input [Varshosaz et al. 2018]. However, our empirical evidence suggests that a three-wise sampling using ICPL already yields a representative subset of configurations for the hypothesis testing. Moreover, the majority of related work only applies pair-wise sampling, which is oftentimes considered to be sufficient for the respective technique [Varshosaz et al. 2018].

## Configurable Verification Systems

Today, a plethora of configurable verification systems exists. For deductive verification, KEY is a program verifier for JAVA programs and was applied successfully to reveal serious defects in deployed code [De Gouw et al. 2015; Mostowski 2005; Mostowski 2007; Ahrendt et al. 2012]. For instance, De Gouw et al. [2015] investigated the correctness of OpenJDK's TimSort with KEY and discovered an exploitable bug in its implementation. Noteworthy, they had to manually change configurations even during the search for proofs, which is difficult as it requires to find meaningful interruption points in a proof search. This is an indicator that complex algorithms require experts

(up-to now). However, the majority of verification tasks can be automated with GUIDO. There exist other configurable deductive program verifiers, such as FRAMA-C [Cuoq et al. 2012] and SPEC# [Barrett et al. 2011]. In the future, we aim at combining these program verifiers with GUIDO as well. In a previous study [Knüppel et al. 2018c], we even suspected that there is a trade-off between verifiability and verification effort for KEY, which we affirmed in **RQ-1**. The long-term goal of these tools aligns with GUIDO, which is to decrease interaction and increase automation in verification.

Another formal verification discipline that deals with huge configuration spaces is model checking. For instance, prominent and configurable model checkers include SPIN [Holzmann 1997] and Java Pathfinder [Havelund et al. 2000]. Moreover, we evaluated CPACHECKER [Beyer et al. 2011] in our final experiments. Our typical experience with model checkers is that most of the time experts are consulted to optimally configure them, whereas our goal with GUIDO is to make verification tools more applicable to practitioners by shipping them with a formalized version of such expert knowledge.

## 6.8. Chapter Summary

In line with the previous chapter, our on-going vision is to *mainstream* deductive verification by supporting developers with adoption in their software engineering practices. To increase automation, we focused on configurable program verifiers and discussed how domain knowledge can improve the underlying proof search. As solution, we presented GUIDO, a tool-agnostic framework that systematically derives adequate configurations based on statistical hypothesis testing.

We implemented GUIDO as a stand-alone open-source tool that can be combined with configurable verification tools targeting imperative programs. Experts need to formalize hypotheses using GUIDO's domain-specific language once. End-users can then apply GUIDO fully automatically.

Our evaluation on KEY-2.7.0 and a suite of 94 verification tasks demonstrated that incorporating domain knowledge leads to better results than KEY's default behavior. Even on a small number of training attempts, GUIDO was able to outperform the default configuration with a random continuation as baseline with respect to verifiability and verification effort.

We believe that our work in this chapter and GUIDO in particular can support three perspectives in the context of formal verification. *Practitioners* are interested in automation and performance, which is the primary goal of GUIDO. *Researchers* are also interested in automation and performance, but additionally focus on data collection, measurements in general, and comparisons against established benchmarks. For instance, many evaluations reporting on improved performances in deductive verification do not report on the applied parameter settings, which impedes reproducibility. Finally, *tool builders* are confronted with numerous questions to improve their tools, such as: *do specific configuration options work as intended? What is the best default configuration? Are there unknown interactions between configuration options or are there unexplainable performance issues?* All these questions are at the focal point of GUIDO.

Based on this chapter, we gained three additional insights. First, understanding the influence of specific parameters is challenging for non-expert users, which we conclude from the fact that many hypotheses we defined could not be accepted. Second, the one-time-effort of formulating the domain knowledge and establishing the benchmark dataset is justified by a significant improvement in effectiveness and performance during the verification process. In particular, recur-

rent re-verification, as performed in the context of continuous integration, will benefit significantly over time. Third, at least for KEY, we identified a trade-off between verifiability and verification effort, which means that specific parameters will either increase success rate or decrease effort, but not both simultaneously. As the field of parameter tweaking in the context of formal verification is currently not well understood, our hope is that GUIDO allows researchers and tool builders to improve this line of research further.

# 7. Conclusion

The importance of cyber-physical systems has significantly increased over the last decades with numerous examples in automotive, robotics, aircraft, or medical applications. High demands on safety requires thorough development processes that ensure functional correctness of such systems to prevent malfunctioning. However, state-of-the-art languages and tools are highly heterogeneous, which complicates system design and quality assurance of new emergent behaviors for developers. Moreover, if formal verification is applied in the process, model checking is still preferred due to automation and the generation of counter examples. In contrast, deductive methods are often regarded as too costly and require more human effort. Still, we argue that certifying the absence of serious defects with formal proofs in the context of cyber-physical systems is indispensable in the near future. In this thesis, we addressed both concerns and presented our vision of a maneuver-centric development process. Our goal was to combine model-based design with deductive verification to (1) derive functional and safe virtual prototypes, (2) maximize reuse of model artifacts and verification results on all levels, and (3) improve applicability of deductive verification, even for non-expert users. We summarize our contributions and conclude this thesis in [Section 7.1](#). In [Section 7.2](#), we discuss potential future work.

## 7.1. Contribution

In the following, we summarize our contributions and aim at answering the three research questions raised in [Chapter 1](#).

**Research Question RQ1 – Modeling and Verifying Maneuvers.** To model and verify the behavior of maneuvers of cyber-physical systems with high abstraction, we presented our modeling approach and tool support SKEDITOR. SKEDITOR successfully combines skill-graph modeling with hybrid programs to specify, model, and verify all parts of a maneuver at design time. The combination and our theoretical considerations turned out to be fruitful, as both concepts prioritize compositionality in their own way. Skill graphs abstract component-based architectures and allow to be developed in isolation. Similarly, hybrid programs allow to describe and verify the controller logic based on the compositional deductive calculus  $d\mathcal{L}$ . *Composition* and *Decomposition* are therefore the primary established properties of our approach. Our evaluation illustrated that reusing already developed parts of a maneuver can therefore decrease verification effort significantly and support practitioners in their development of new maneuvers.

**Research Question RQ2 – Architectural Refinement and Simulation.** The second question focused on deriving a functional and virtual prototype by refining skill graphs to software components, which can eventually be simulated within a computer. Due to our highly abstract modeling approach, it is necessary to complete parts of the implementation manually. We introduced ARCHICORC,

which lifts the correctness-by-construction paradigm to component-based architectures and provides means for automatic code generation. Our derived guidelines show that – in principle – correctness-by-construction can be used to complete all remaining parts without introducing new defects to the implementation. Except for the fact that we have not formally verified correctness of our code generation process, using correctness-by-construction this way allows us to conclude that the implementation’s behavior is a valid refinement of the modeled behavior. This means that violations during simulation are almost certainly due to incomplete requirements and not caused by defective modeling or faulty implementations. Indeed, in our evaluation, all safety properties verified at design time also hold during simulation.

**Research Question RQ<sub>3</sub> – Mainstreaming Deductive Verification.** Although broadly stated, the third question specifically addresses improvements in the context of deductive verification and verification tools at implementation level. The two challenges we focused on were (1) assessing completeness of software contracts and (2) studying the parameter space of configurable program verifiers. For the first challenge, we studied to what extent a mutation analysis can provide insights on incomplete contracts. Despite the simplicity of this approach, an interesting insight is that manual inspection of many alive mutants quickly led to the cause of incompleteness. Whereas the mutation score can sometimes be misleading with respect to relative incompleteness, each alive mutant constitutes a counter argument to a contract’s completeness and can further be analyzed.

For the second challenge, we introduced GUIDO and studied the parameter space of KEY 2.7.0 and CPACHECKER 1.8. We propose to combine explicit domain knowledge in the form of statistical hypotheses to (1) invalidate false assumptions about specific parameters and (2) overcome the limitations of general-purpose prediction tools in the context of deductive verification (e.g., number of measurements needed). We believe that new emerging program verifiers will still base their algorithms on numerous control parameters, either implicitly and hidden from end-user, or explicitly through the user interface. We are confident that GUIDO can serve as a benchmark tool for tool builders, but also support non-expert users in configuring their verification tool of choice.

## 7.2. Future Work

The presented development process for maneuver-based cyber-physical systems and the integration of deductive verification as presented in this thesis offers many prospects for future work. We present some of the most interesting directions in the following.

**Modeling and Verifying Non-Functional Properties.** Although important properties in the context of cyber-physical systems, we did not consider non-functional requirements and uncertainty. In this direction, enabling *redundant* modeling of functionality and sensors based on uncertainty measurements would increase robustness of real-world application of skill graphs. For instance, we assumed ideal values from our sensors in our verification process, whereas sensor quality depends on numerous factors, such as employed hardware and weather conditions. Adding means for specifying such assumptions allows developers to add redundant sensors and perception units in case of degraded functionality. Consequently, different skill graphs can be applied in different scenarios to



optimize resource consumption based on external conditions. Another important non-functional property for optimizing resource consumption is (worst-case) execution time. Adding timing information to both skills in SKEDITOR and components in ARCHICORC enables assume-guarantee reasoning for the timing viewpoint. This prevents the development of completely unoptimized virtual prototypes and increases the chances of using such prototypes in real-world operation as-is.

**Augmentation of Skill Graphs for more Analyses.** The presented skill-graph models can be augmented with additional information to widen the scope of applicable analyses. Whereas our focal point lied on formal verification only, other static analyses can improve the general development process for practitioners. For instance, applicability of skill graphs for virtual prototypes depends on the used simulation models, which we knew conformed to each other in our evaluations. For instance, our presented skill graphs for the robotic domain are not applicable in the automotive domain. With enough information, identifying conformance of skill graphs and simulation model can be automated. Other examples for static analyses include decomposability through slicing (*can I split a skill graph into smaller skill graphs?*), embedding (*is one skill graph embedded in another skill graph?*), and complexity measurements (*is the skill graph minimal or can it be reduced further?*).

**Synthesis of Skill Graphs.** Developing complex skill graphs from scratch can require high human effort and may need a lot of expertise. The structure of skill graphs, however, may allow to synthesize more complex skill graphs from simpler ones. That is, skills and (partial) skill graphs can be viewed in isolation and are largely described by their specification. A future goal is to establish a *repository* of skills and skill graphs that can be reused and also used as input for synthesis algorithms. Starting with a set of requirements provided by the modeler, missing parts of a skill graph can then be completed automatically by considering their specification. In case of ambiguity, a semi-automatic approach may suggest a number of alternatives and guide modelers through the completion process. Part of this future work is also to identify how a query language for modelers may look like to derive a skill graph representing the maneuver they have in mind.

**Language and Tool Adoption.** Integration of additional specification languages, programming languages, and employed verification tools can be improved. Whereas there exist little competition to hybrid programs in the area of deductive verification for cyber-physical systems, there exist numerous alternatives for deductive program verifiers on source-code level, such as FRAMA-C [Cuoq et al. 2012] and SPEC# [Barnett et al. 2011]. It is particularly important to reproduce the empirical evaluations of this thesis on other languages to examine generalization of our gained insights.

**Intersection of Software Engineering and Formal Methods.** Besides specification completeness and parameter configuration of formal verifiers, there exist other interesting challenges to address that will increase the adoption of deductive verification in software engineering processes. For instance, specified methods of object-oriented programs that are called by another method can be either inlined or abstracted with their respective contract during verification. Although contract abstraction is sometimes unavoidable (e.g., in case of missing implementation), there exist reasons to believe that inlining can be more cost-effective in specific instances [Knüppel et al. 2018b]. It is therefore reasonable to study prediction models for deciding automatically when to inline an imple-

mentation or when to use its contract. Another important problem in continuous integration is the evolution of software and the corresponding invalidation of proofs. For a successful adoption of deductive verification, proofs should not be discarded, but rather be *patched* in accordance to the evolved software. This requires sophisticated research on *proof reuse*, *proof repair*, and potentially affects even the underlying language of the *proof format*.

**Employing Model Checking and Theorem Proving in Concert.** Our primary focus was on theorem proving and deductive verification to generate proofs that certify the correctness of the intended behavior. However, theorem proving works best if specification and behavior indeed conform. If they diverge, practitioners of all levels of expertise can easily become confused with the results so far. We acknowledge that integrating model checking early on in the process to identify defects in model and implementation enhances the debugging process significantly. For instance, it is imaginable to employ theorem proving for certification only after no more counterexamples are producible by an employed model checker.

# Appendix



# A. Results of the Mutation Analysis

In our evaluation in [Section 5.4](#), we only showed the averaged mutation scores per project. In the following, we present the obtained results per analyzed method. Equivalent mutants are excluded. Therefore, valid mutants as considered in the following tables are non-equivalent to the source implementation. Moreover, no mutants are generated for methods that cannot be parsed or otherwise raise an exception. In these cases, we rate the respective method with a mutation score of 0.00. The studies and the prototypical tool support are available in our online repository.<sup>1</sup>

## Statistics on Source-Code Mutation

The results of our empirical evaluation of source-code mutation for contract-based software are shown in [Table A.1](#). We illustrate (1) the number of total mutants, and (2) the number of valid mutants, and (3) the resulting mutation score.

Project	Class	Method	#Mutants	#Verified	#Score
Dnvira	Account	int_getBalance()	5	0	1.00
	Account	void_depositAmount(int)	10	0	1.00
	Account	void_withdrawAmount(int)	10	0	1.00
SchorrWaiteAlgorithm	HeapObject	boolean_hasNext()	25	0	1.00
	HeapObject	boolean_isMarked()	3	0	1.00
	HeapObject	HeapObject_getChild(int)	2	0	1.00
	HeapObject	int_getChildCount()	5	0	1.00
	HeapObject	int_getIndex()	5	0	1.00
	HeapObject	void_incIndex()	5	0	1.00
	HeapObject	void_setMark(boolean)	3	0	1.00
PayCard	LogRecord	int_getTransactionID()	0	0	1.00
	LogRecord	int_getBalance()	5	0	1.00
	LogRecord	void_setRecord(int)	31	5	0.83
	PayCard	PayCard_createJuniorCard()	0	0	1.00
	PayCard	boolean_isValid()	20	0	1.00
	PayCard	boolean_charge(int)	76	0	1.00
BankAccount	Account	int_checkAndWithdraw(int)	0	0	0.00
	Account	int_getAccountNumber()	0	0	1.00
	Account	void_addTransaction(Transaction)	0	0	0.00
	ATM	boolean_proxyExists(int)	10	0	1.00
	ATM	OfflineAccountProxy_getProxy(int)	2	0	1.00
	ATM	void_insertCard(BankCard)	0	0	0.00
	BankCard	boolean_cardIsInvalid()	3	0	1.00
	BankCard	boolean_pinIsCorrect()	27	0	1.00
	BankCard	int_getAccountNumber()	5	0	1.00
	BankCard	void_makeCardInvalid()	2	0	1.00
	CentralHost	BankCard_issueCard(int,int)	4	0	1.0
	CentralHost	boolean_accountExists(int)	13	1	0.92
	CentralHost	PermanentAccount_getAccount(int)	0	0	0
	CentralHost	void_createAccount(int)	8	8	0.00
	Clock	clock_getInstance()	0	0	1.00
	OfflineAccountProxy	boolean_newWithdrawalIsPossible(int)	26	18	0.30

<sup>1</sup>Available at <https://github.com/TUBS-ISF/MutationAnalysisForDBC-Formalise21>.

	OfflineAccountProxy	int_accountBalance()	1	1	0.00
	PermanentAccount	boolean_dailyLimitIsImportant(int)	31	31	0.00
	PermanentAccount	int_accountBalance()	5	3	0.40
BankAccountV2	Account	boolean_undoUpdate(int)	73	1	0.98
	Account	boolean_update(int)	73	1	0.98
	Account	int_estimatedInterest(int)	26	3	0.88
	Account	int_calculateInterest()	27	13	0.51
	Transaction	boolean_lock(Account,Account)	9	5	0.44
	Transaction	boolean_transfer(Account,Account,int)	53	1	0.98
PayCardSPL	LogRecord	int_getBalance()	5	0	1.00
	LogRecord	int_getTransactionID()	5	0	1.00
	LogRecord	void_setRecord(int)	32	6	0.81
	PayCard	PayCard_createJuniorCard()	0	0	1.0
	PayCard	boolean_charge__wrappee__Paycard(int)	71	0	1.00
ExamplesFromChapter7	ArraySearchWhile	boolean_search(int,int)	56	0	1.00
	ArraySumForEach	int_sum(int)	15	0	1.00
	StudentA	String_getName()	0	0	1.00
	StudentA	void_addCredits(int)	22	6	0.72
	StudentA	void_changeToMaster()	2	0	1.00
	StudentA	void_setName(String)	0	0	1.00
	StudentA	void_updateCredits(int)	10	0	1.00
ExamplesFromChapter16	PostInc	void_postinc()	7	0	1.00
	Sort	int_max(int)	80	0	1.00
	Sort	void_sort()	65	1	0.98
OpenJML	Bag	void_add(int)	66	35	0.46
	BeanCan	boolean_pick_random()	0	0	1.00
	BeanCan	void_remove(boolean)	8	0	1.00
	BeanCan	boolean_play_game()	6	0	1.00
	CashAmount	int_cents()	5	0	1.00
	CashAmount	int_dollars()	5	0	1.00
	CashAmount	CashAmount_negate()	8	0	1.00
	Customer	void_enter()	1	0	1.00
	Customer	void_leave()	2	0	1.00
	Customer	void_request()	9	0	1.00
	LoopExamples	void_setA(int)	1	1	0.00
	MaxByElimination	int_max(int)	74	0	1.00
	TickTockClock	int_getHour()	5	1	0.80
	TickTockClock	int_getMinutes()	5	1	0.80
	TickTockClock	int_getSeconds()	5	1	0.80
	TickTockClock	void_tick()	72	4	0.94
	Time	byte_getHour()	4	0	1.00
	Time	byte_getMinute()	4	0	1.00
	Time	short_getTime(byte,short)	37	16	0.56
	Time	void_setTime(byte,byte)	10	0	1.00
DutchFlagAlgorithm	Debug2	int_DutchFlag(int)	181	2	0.99

Table A.1.: Mutation scores per method and project for source-code mutation.

### Statistics on Contract-Level Mutation

The results of our empirical evaluation of contract-level mutation (i.e., operators PW and PS) for contract-based software are shown in [Table A.2](#). We illustrate (1) the number of total mutants, and (2) the number of valid mutants produced by either operator PW or operator PS.

Project	Class	Method	#Mutants	#PW	#PS
Dnvira	Account	int_getBalance()	0	0	0
	Account	void_depositAmount(int)	2	1	0
	Account	void_withdrawAmount(int)	4	2	0
SchorrWaiteAlgorithm	HeapObject	boolean_hasNext()	1	0	0
	HeapObject	boolean_isMarked()	0	0	0
	HeapObject	HeapObject_getChild(int)	7	0	0
	HeapObject	int_getChildCount()	0	0	0

	HeapObject	int_getIndex()	0	0	0
	HeapObject	void_incIndex()	0	0	0
	HeapObject	void_setMark(boolean)	0	0	0
PayCard	LogRecord	int_getTransactionID()	0	0	0
	LogRecord	int_getBalance()	0	0	0
	LogRecord	void_setRecord(int)	5	0	1
	PayCard	PayCard_createJuniorCard()	0	0	0
	PayCard	boolean_isValid()	2	0	0
	PayCard	boolean_charge(int)	5	0	0
BankAccount	Account	int_checkAndWithdraw(int)	3	0	0
	Account	int_getAccountNumber()	0	0	0
	Account	void_addTransaction(Transaction)	0	0	0
	ATM	boolean_proxyExists(int)	3	1	0
	ATM	OfflineAccountProxy_getProxy(int)	3	0	0
	ATM	void_insertCard(BankCard)	3	3	0
	BankCard	boolean_cardIsInvalid()	0	0	0
	BankCard	boolean_pinIsCorrect()	0	0	0
	BankCard	int_getAccountNumber()	0	0	0
	BankCard	void_makeCardInvalid()	0	0	0
	CentralHost	BankCard_issueCard(int,int)	0	0	0
	CentralHost	boolean_accountExists(int)	3	0	0
	CentralHost	PermanentAccount_getAccount(int)	0	0	0
	CentralHost	void_createAccount(int)	3	0	0
	Clock	clock_getInstance()	0	0	0
	OfflineAccountProxy	boolean_newWithdrawalIsPossible(int)	2	2	0
	OfflineAccountProxy	int_accountBalance()	0	0	0
	PermanentAccount	boolean_dailyLimitIsImportant(int)	0	0	0
	PermanentAccount	int_accountBalance()	0	0	0
BankAccountV2	Account	boolean_undoUpdate(int)	0	0	0
	Account	boolean_update(int)	0	0	0
	Account	int_estimatedInterest(int)	4	1	0
	Account	int_calculateInterest()	2	0	0
	Transaction	boolean_lock(Account,Account)	1	1	0
	Transaction	boolean_transfer(Account,Account,int)	1	0	0
PayCardSPL	LogRecord	int_getBalance()	0	0	0
	LogRecord	int_getTransactionID()	0	0	0
	LogRecord	void_setRecord(int)	1	0	0
	PayCard	PayCard_createJuniorCard()	0	0	0
	PayCard	boolean_charge__wrappee__Paycard(int)	9	0	0
ExamplesFromChapter7	ArraySearchWhile	boolean_search(int,int)	3	0	0
	ArraySumForEach	int_sum(int)	0	0	0
	StudentA	String_getName()	0	0	0
	StudentA	void_addCredits(int)	1	0	0
	StudentA	void_changeToMaster()	1	1	0
	StudentA	void_setName(String)	0	0	0
	StudentA	void_updateCredits(int)	1	0	0
ExamplesFromChapter16	PostInc	void_postinc()	0	0	0
	Sort	int_max(int)	10	1	0
	Sort	void_sort()	2	0	0
OpenJML	Bag	void_add(int)	0	0	0
	BeanCan	boolean_pick_random()	0	0	0
	BeanCan	void_remove(boolean)	0	0	0
	BeanCan	boolean_play_game()	0	0	0
	CashAmount	int_cents()	0	0	0
	CashAmount	int_dollars()	0	0	0
	CashAmount	CashAmount_negate()	0	0	0
	Customer	void_enter()	3	1	0
	Customer	void_leave()	2	2	0
	Customer	void_request()	2	2	0
	LoopExamples	void_setA(int)	0	0	0
	MaxByElimination	int_max(int)	4	0	0
	TickTockClock	int_getHour()	4	0	0
	TickTockClock	int_getMinutes()	4	0	0



	TickTockClock	int_getSeconds()	4	0	0
	TickTockClock	void_tick()	4	0	0
	Time	byte_getHour()	0	0	0
	Time	byte_getMinute()	0	0	0
	Time	short_getTime(byte,short)	8	8	0
	Time	void_setTime(byte,byte)	10	4	0
DutchFlagAlgorithm	Debugz	int_DutchFlag(int)	12	2	0

Table A.2.: Mutation scores per method and project for contract-level mutation.

# B. Supplemental Material for GUIDO

In [Chapter 6](#), we explained how GUIDO incorporates explicit domain knowledge in the form of statistical hypotheses. In the following, we illustrate the considered parameter space, the hypotheses we formulated, and which hypotheses could be accepted for KEY-2.7.0 and CPACHECKER-1.8. The formulated domain knowledge and additional data (e.g., benchmark dataset for both KEY and CPACHECKER, and R-scripts) are available in our online repository.<sup>1</sup>

## B.1. Hypotheses for KEY-2.7.0

In total, we formulated 47 assumptions about verifiability and verification effort. Each assumption targets one parameter of KEY. As parameters of KEY may comprise more than two options, an assumption is potentially split into more than one hypothesis. The translation of the 47 assumptions therefore results in 72 statistical and testable hypotheses. Each statistical hypothesis is investigated as one independent experiment.

The hypotheses and resulting experiments are shown in [Table B.1](#). The first column assigns a unique number to each assumption. The third column defines the statistical hypothesis number (i.e., the experiment). Numbers are assigned in chronological order. The parameter is shown in the second column and the compared configuration options are shown in the fourth and fifth column. As we formulated hypotheses about verifiability (P) and verification effort (VE), the regarding requirement of a hypothesis is shown in column six. The direction of improvement is encoded in column seven. The dependency  $<>$  states that there is no difference between the first second option, the dependency  $\geq$  states that the first option is likely to either improve verifiability or impair verification effort compared to the second option. The dependency  $\leq$  is used for the opposite. For example, Hypothesis 57 states that option *Stop At::Default* impairs the verification effort compared to option *Stop At::Unclosable*. Finally, the p-value is shown in bold when the hypothesis was accepted with respect to the benchmark dataset in use.

---

<sup>1</sup>Available at <https://github.com/AlexanderKnueppel/Guido>.

Assumption	Parameter	Hypothesis	First Option	Second Option	Requirement	Dependency	p-value
1	Stop At	1	Default	Unclosable	P	$\diamond$	0.95
2	Stop At	2	Default	Unclosable	VE	$\diamond$	0.49
41	Stop At	57	Default	Unclosable	VE	$\geq$	0.54
3	One Step Simplification	3	Enabled	Disabled	P	$\diamond$	<b>4.0e-3</b>
4	One Step Simplification	4	Enabled	Disabled	VE	$\leq$	<b>&lt; 2.2e-16</b>
5	Proof Splitting	5	Delayed	Free	P	$\leq$	N/A
6	Proof Splitting	6	Delayed	Free	VE	$\leq$	<b>5.0e-4</b>
7	Proof Splitting	7	Off	Free	P	$\leq$	<b>&lt; 2.2e-16</b>
		8	Off	Delayed	P	$\leq$	<b>&lt; 2.2e-16</b>
8	Proof Splitting	9	Off	Free	VE	$\leq$	0.61
		10	Off	Delayed	VE	$\leq$	0.88
9	Loop Treatment	11	Invariant	Loop Scope Invariant	P	$\leq$	N/A
10	Loop Treatment	12	Invariant	Loop Scope Invariant	VE	$\geq$	1
11	Dependency Contracts without <b>accessible</b> -Clauses	13	On	Off	P	$\diamond$	0.25
12	Dependency Contracts without <b>accessible</b> -Clauses	14	On	Off	VE	$\diamond$	0.54
13	Query Treatment without Queries	15	On	Restricted	P	$\diamond$	N/A
		16	On	Off	P	$\diamond$	N/A
14	Query Treatment without Queries	17	On	Restricted	VE	$\diamond$	N/A
		18	On	Off	VE	$\diamond$	N/A
15	Query Treatment	19	Off	Restricted	P	$\leq$	N/A
		20	Restricted	On	P	$\leq$	N/A
16	Query Treatment	21	Restricted	On	VE	$\diamond$	1
17	Expand Local Queries	22	On	Off	VE	$\geq$	<b>2.6e-06</b>
18	Expand Local Queries	23	On	Off	P	$\geq$	N/A
19	Arithmetic Treatment	24	Basic	DefOps	P	$\leq$	<b>&lt; 2.2e-16</b>
20	Arithmetic Treatment	25	DefOps	ModelSearch	P	$\diamond$	0.24
39	Arithmetic Treatment	54	Basic	ModelSearch	P	$\leq$	<b>&lt; 2.2e-16</b>
42	Arithmetic Treatment	58	DefOps	ModelSearch	VE	$\geq$	<b>6.6e-4</b>
		59	Basic	ModelSearch	VE	$\geq$	1
47	Arithmetic Treatment with Method Calls without own contract	71	ModelSearch	Basic	P	$\leq$	N/A
		72	ModelSearch	DefOps	P	$\leq$	N/A
21	Quantifier Treatment without Quantifiers	26	None	No Splits	P	$\diamond$	<b>5.7e-4</b>
		27	None	No Splits With Progs	P	$\diamond$	0.14
		28	None	Free	P	$\diamond$	8.7e-4
22	Quantifier Treatment without Quantifiers	29	None	No Splits	VE	$\diamond$	<b>8.7e-4</b>
		30	None	No Splits With Progs	VE	$\diamond$	N/A
		31	None	Free	VE	$\diamond$	0.15
23	Quantifier Treatment	32	None	No Splits	P	$\leq$	<b>5.7e-4</b>
		33	No Splits	No Splits With Progs	P	$\leq$	0.17
		34	No Splits With Progs	Free	P	$\leq$	0.22
24	Quantifier Treatment	35	Free	No Splits With Progs	VE	$\geq$	<b>6.4e-08</b>
		36	No Splits With Progs	No Splits	VE	$\geq$	1
		37	No Splits	None	VE	$\geq$	0.92
25	Class Axiom Rule without Axioms	38	Free	Delayed	P	$\diamond$	N/A
		39	Free	Off	P	$\diamond$	N/A
26	Class Axiom Rule without Axioms	40	Free	Delayed	VE	$\diamond$	N/A
		41	Free	Off	VE	$\diamond$	N/A
27	Class Axiom Rule	52	Considered separately				
28	Class Axiom Rule	53	Off	Delayed	P	$\diamond$	N/A
40	Class Axiom Rule	55	Off	Delayed	P	$\leq$	<b>&lt; 2.2e-16</b>
		56	Off	Free	P	$\leq$	<b>&lt; 2.2e-16</b>
43	Class Axiom Rule	60	Delayed	Free	VE	$\geq$	1
		61	Delayed	Off	VE	$\geq$	1

44	Class Axiom Rule with Implications	62	Off	Free	P	$\diamond$	N/A
		63	Delayed	Free	P	$\diamond$	N/A
		64	Off	Delayed	P	$\diamond$	N/A
45	Class Axiom Rule without Implications	65	Delayed	Free	P	$\diamond$	N/A
		66	Off	Delayed	P	$\leq$	<b>&lt; 2.2e-16</b>
		67	Off	Free	P	$\leq$	N/A
46	Class Axiom Rule with Conditional Statements	68	Delayed	Free	P	$\diamond$	N/A
		69	Off	Delayed	P	$\leq$	<b>&lt; 2.2e-16</b>
		70	Off	Free	P	$\leq$	N/A
29	Strings	42	On	Off	P	$\geq$	N/A
30	Strings	43	On	Off	VE	$\diamond$	N/A
31	BigInt	44	On	Off	P	$\geq$	0.89
32	BigInt	45	On	Off	VE	$\diamond$	0.51
33	IntegerSimplificationRules	46	Full	Minimal	P	$\geq$	<b>4.2e-8</b>
34	IntegerSimplificationRules	47	Full	Minimal	VE	$\diamond$	0.668
35	Sequences	48	On	Off	P	$\geq$	3.6e.3
36	Sequences	49	On	Off	VE	$\diamond$	0.51
37	MoreSeqRules	50	On	Off	P	$\geq$	N/A
38	MoreSeqRules	51	On	Off	VE	$\diamond$	N/A

Table B.1.: Formulated Hypotheses and Experiments

## B.2. Hypotheses for CPACHECKER-4.9.0

For CPACHECKER, we formulated a total of ten assumptions, resulting in 14 independent experiences. Table B.2 shows the results (cf. autorefappendix:guido:key for a description of the table's columns). Based on our benchmark dataset, we could except eleven hypotheses.

Assumption	Parameter	Hypothesis	First Option	Second Option	Requirement	Dependency	p-value
1	analysis.summaryEdges	H1	true	false	P	$\geq$	<b>&lt; 2.2e-16</b>
2	analysis.traversal.order	H2	dfs	bfs	VE	$\leq$	0.05
		H3	dfs	random_path	VE	$\leq$	0.06
3	analysis.useParallelAnalyses	H4	true	false	P	$\leq$	<b>3.2e-11</b>
4	analysis.useParallelAnalyses	H5	false	true	VE	$\geq$	<b>&lt; 2.2e-16</b>
5	analysis.useParallelAnalyses	H6	false	true	VE	$\geq$	<b>0.01</b>
6	cpa.invariants.abstractionStateFactory	H7	entering_edges	always	VE	$\leq$	<b>0.02</b>
		H8	always	never	VE	$\leq$	<b>0.02</b>
7	cpa.smg.exportSMGWhen	H9	interesting	every	VE	$\leq$	<b>0.01</b>
		H10	never	every	VE	$\geq$	<b>0.01</b>
		H11	never	leave	VE	$\geq$	<b>0.02</b>
8	cpa.smg.memoryErrors	H12	true	false	P	$\geq$	0.06
9	cpa.smg.unknownOnUndefined	H13	false	true	P	$\leq$	<b>0.03</b>
10	cpa.predicate.handleStringLiteralInitializers	H14	true	false	P	$\leq$	<b>0.02</b>

Table B.2.: Formulated Hypotheses and Experiments



# Bibliography

- Abbasi, R., J. Schiffl, E. Darulova, M. Ulbrich, and W. Ahrendt (2021). “Deductive Verification of Floating-Point Java Programs in KeY.” In: *Proc. of the Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 242–261.
- Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.
- Abrial, J.-R., M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin (2010). “Rodin: an open toolset for modelling and reasoning in Event-B”. *International Journal on Software Tools for Technology Transfer* 12.6, pp. 447–466.
- Agda. *Agda Development Team. The Agda wiki, 2007-2021*. <http://wiki.portal.chalmers.se/agda/pmwiki.php>. Accessed: 2021-06-18.
- Ahmed, B. S., K. Z. Zamli, W. Afzal, and M. Bures (2017). “Constrained interaction testing: A systematic literature study”. *IEEE Access* 5, pp. 25706–25730.
- Ahrendt, W., B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich (2016). *Deductive software verification – The KeY book*. Vol. 10001. Springer.
- Ahrendt, W., W. Mostowski, and G. Paganelli (2012). “Real-time Java API specifications for high coverage test generation”. In: *Proc. of the Intl. Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. ACM, pp. 145–154.
- Alur, R. (2011). “Formal verification of hybrid systems”. In: *Proc. of the Intl. Conference on Embedded Software and Systems*, pp. 273–278.
- Alur, R. (2015). *Principles of Cyber-Physical Systems*. The MIT Press.
- Alur, R., C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine (1995). “The algorithmic analysis of hybrid systems”. *Theoretical Computer Science* 138.1, pp. 3–34.
- Alur, R., C. Courcoubetis, T. A. Henzinger, and P.-H. Ho (1992). “Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems”. In: *Hybrid systems*. Springer, pp. 209–229.
- Alur, R., T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivani, V. Kumar, I. Lee, P. Mishra, G. Pappas, et al. (2001). “Hierarchical hybrid modeling of embedded systems”. In: *Proc. of the Intl. Workshop on Embedded Software (EMSOFT)*. Springer, pp. 14–31.
- Andrews, J. H., L. C. Briand, and Y. Labiche (2005). “Is mutation an appropriate tool for testing experiments?” In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*, pp. 402–411.

- Angermann, A., M. Beuschel, M. Rau, and U. Wohlfarth (2020). *Matlab–simulink–stateflow*. De Gruyter Oldenbourg.
- Atkinson, C., P. Bostan, D. Brenner, G. Falcone, M. Gutheil, O. Hummel, M. Juhasz, and D. Stoll (2008). “Modeling components and component-based systems in Kobra”. In: *The Common Component Modeling Example*. Springer, pp. 54–84.
- Back, R.-J. (2009). “Invariant based programming: basic approach and teaching experiences”. *Formal Aspects of Computing* 21.3, pp. 227–244.
- Back, R.-J., J. Eriksson, and M. Myreen (2007). “Testing and verifying invariant based programs in the SOCOS environment”. In: *Proc. of the Intl. Conference on Tests and Proof (TAP)*. Springer, pp. 61–78.
- Bagschik, G., M. Nolte, S. Ernst, and M. Maurer (2018). “A system’s perspective towards an architecture framework for safe automated vehicles”. In: *Proc. of the Intl. Conference on Intelligent Transportation Systems (ITSC)*. IEEE, pp. 2438–2445.
- Bak, S., S. Bogomolov, and T. T. Johnson (2015). “HYST: a source transformation and translation tool for hybrid automaton models”. In: *Proc. of the Intl. Conference on Hybrid Systems: Computation and Control (HSCC)*, pp. 128–133.
- Ball, T. and O. Kupferman (2008). “Vacuity in testing”. In: *Proc. of the Intl. Conference on Tests and Proof (TAP)*. Springer, pp. 4–17.
- Banach, R. and M. Poppleton (1998). “Retrenchment: An Engineering Variation on Refinement”. In: *Proc. of the Intl. Conference of B Users*. Springer, pp. 129–147.
- Baracchi, L., A. Cimatti, G. Garcia, S. Mazzini, S. Puri, and S. Tonetta (2014). “Requirements refinement and component reuse: the FoReVer contract-based approach”. In: *Handbook of Research on Embedded Systems Design*. IGI Global, pp. 209–241.
- Barnes, J. (2012). *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis.
- Barnett, M., M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter (June 2011). “Specification and Verification: The Spec# Experience”. *CACM* 54 (6), pp. 81–91. DOI: <http://doi.acm.org/10.1145/1953122.1953145>.
- Barth, T., M. P. McKay, and R. Smith (2020). “Survivability of a Tesla collision into a non-operational crash attenuator in Mountain View, CA”. *Traffic injury prevention*, pp. 1–3.
- Baudin, P., J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto (2008). “ACSL: Ansi C specification language”. *CEA-LIST, Saclay, France, Tech. Rep. v1 2*.
- Bauer, S. S., A. David, R. Hennicker, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski (2012). “Moving from specifications to contracts in component-based design”. In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, pp. 43–58.
- Baumann, C., B. Beckert, H. Blasum, and T. Bormer (2012). “Lessons learned from microkernel verification: specification is the new bottleneck”. In: *Proc. of the Conference on Systems Software Verification (SSV)*, pp. 18–32.



- Becker, S., H. Koziol, and R. Reussner (2009). "The Palladio component model for model-driven performance prediction". *Journal of Systems and Software* 82.1, pp. 3–22.
- Beckert, B. and V. Klebanov (2004). "Proof reuse for deductive program verification". In: *Proc. of the Intl. Conference on Software Engineering and Formal Methods (SEFM)*. IEEE, pp. 77–86.
- Beckert, B. and S. Schlager (2005). "Refinement and Retrenchment for Programming Language Data Types". *Formal Aspects of Computing* 17.4, pp. 423–442.
- Belli, E. (2018). "Implementing a Graphical Skill Graph Editor for Monitoring Vehicle Guidance Systems". BA thesis. Technical University of Braunschweig.
- Benesty, J., J. Chen, Y. Huang, and I. Cohen (2009). "Pearson correlation coefficient". In: *Noise reduction in speech processing*. Springer, pp. 1–4.
- Benveniste, A., B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis (2007). "Multiple Viewpoint Contract-Based Specification and Design". In: *Proc. of the Intl. Symposium on Formal Methods for Components and Objects (FMCO)*. Springer, pp. 200–225.
- Benveniste, A., B. Caillaud, and R. Passerone (2009). "Multi-Viewpoint State Machines for Rich Component Models". *Model-Based Design of Heterogeneous Embedded Systems*.
- Benveniste, A., B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, K. G. Larsen, et al. (2018). "Contracts for system design". *Foundations and Trends® in Electronic Design Automation*.
- Benvenuti, L., D. Bresolin, P. Collins, A. Ferrari, L. Geretti, and T. Villa (2014). "Assume–guarantee verification of nonlinear hybrid systems with Ariadne". *International Journal of Robust and Nonlinear Control* 24.4, pp. 699–724.
- Bergmiller, P. J. (2015). *Towards functional safety in drive-by-wire vehicles*. Springer.
- Bernardi, S., U. Gentile, S. Marrone, J. Merseguer, and R. Nardone (2021). "Security modelling and formal verification of survivability properties: Application to cyber–physical systems". *Journal of Systems and Software* 171, p. 110746.
- Bernardi, S. and J. Merseguer (2007). "A UML profile for dependability analysis of real-time embedded systems". In: *Proc. of the Intl. Workshop on Software and Performance (WOSP)*, pp. 115–124.
- Beyer, D. and M. E. Keremoglu (2011). "CPAchecker: A tool for configurable software verification". In: *Proc. of the Intl. Conference on Computer Aided Verification (CAV)*. Springer, pp. 184–190.
- Biere, A., A. Cimatti, E. Clarke, and Y. Zhu (1999). "Symbolic model checking without BDDs". In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer, pp. 193–207.
- Biere, A., A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu (2003). "Bounded model checking".
- Biggerstaff, T. J. (1994). "The library scaling problem and the limits of concrete component reuse". In: *Proc. of the Intl. Conference on Software Reuse (ICSR)*. IEEE, pp. 102–109.

- Bledsoe, W. W. (2020). *Some thoughts on proof discovery*. CRC Press.
- Bohrer, B., Y. K. Tan, S. Mitsch, M. O. Myreen, and A. Platzer (2018). “VeriPhy: Verified controller executables from verified cyber-physical system models”. In: *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, pp. 617–630.
- Bordis, T., T. Runge, A. Knüppel, T. Thüm, and I. Schaefer (2020). “Variational correctness-by-construction”. In: *Proc. of the International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS)*. Ed. by M. Cordy, M. Acher, D. Beuche, and G. Saake. ACM, 7:1–7:9. DOI: [10.1145/3377024.3377038](https://doi.org/10.1145/3377024.3377038). URL: <https://doi.org/10.1145/3377024.3377038>.
- Börger, E. (2010). “The abstract state machines method for high-level system design and analysis”. In: *Formal Methods: State of the Art and New Directions*. Springer, pp. 79–116.
- Brambilla, M., J. Cabot, and M. Wimmer (2017). “Model-driven software engineering in practice”. *Synthesis Lectures on Software Engineering* 3.1, pp. 1–207.
- Branicky, M. S. (2005). “Introduction to hybrid systems”. In: *Handbook of networked and embedded control systems*. Springer, pp. 91–116.
- Broy, M., F. Huber, and B. Schätz (1999). “AutoFocus—Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme”. *Informatik Forschung und Entwicklung* 14.3, pp. 121–134.
- Bubel, R., F. Damiani, R. Hähnle, E. B. Johnsen, O. Owe, I. Schaefer, and I. C. Yu (2016). “Proof repositories for compositional verification of evolving software systems”. In: *Transactions on Foundations for Mastering Change I*. Springer, pp. 130–156.
- Bubel, R., R. Hähnle, and M. Plevina (2014). “Fully abstract operation contracts”. In: *Proc. of the Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Springer, pp. 120–134.
- Budd, T. A., R. A. DeMillo, R. J. Lipton, and F. G. Sayward (1980). “Theoretical and empirical studies on using program mutation to test the functional correctness of programs”. In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*, pp. 220–233.
- Buzdalov, D. and A. Khoroshilov (2014). “A discrete-event simulator for early validation of avionics systems”. In: *AProc. of the Workshop on Architecture Centric Virtual Integration (ACVIP)*, p. 28.
- Calder, M., M. Kolberg, E. H. Magill, and S. Reiff-Marganiec (2003). “Feature interaction: a critical review and considered forecast”. *Computer Networks* 41.1, pp. 115–141.
- Celik, A., K. Palmskog, M. Parovic, E. J. G. Arias, and M. Gligoric (2019). “Mutation analysis for Coq”. In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, pp. 539–551.
- Chalin, P., J. R. Kiniry, G. T. Leavens, and E. Poll (2005). “Beyond assertions: Advanced specification and verification with JML and ESC/Java2”. In: *Proc. of the Intl. Symposium on Formal Methods for Components and Objects (FMCO)*. Springer, pp. 342–363.
- Chang, C.-L. and R. C.-T. Lee (2014). *Symbolic logic and mechanical theorem proving*. Academic press.

- Chaochen, Z., W. Ji, and A. P. Ravn (1995). “A formal description of hybrid systems”. In: *Proc. of the Intl. Hybrid Systems Workshop*. Springer, pp. 511–530.
- Chen, X., E. Ábrahám, and S. Sankaranarayanan (2013). “Flow\*: An analyzer for non-linear hybrid systems”. In: *Proc. of the Intl. Conference on Computer Aided Verification (CAV)*. Springer, pp. 258–263.
- Chockler, H., O. Kupferman, and M. Y. Vardi (2003). “Coverage metrics for formal verification”. In: *Proc. of the Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, pp. 111–125.
- Cimatti, A., E. Clarke, F. Giunchiglia, and M. Roveri (1999). “NuSMV: A new symbolic model verifier”. In: *Proc. of the Intl. Conference on Computer Aided Verification (CAV)*. Springer, pp. 495–499.
- Cimatti, A., M. Dorigatti, and S. Tonetta (2013). “OCRA: A tool for checking the refinement of temporal contracts”. In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, pp. 702–705.
- Cimatti, A. and S. Tonetta (2012). “A property-based proof system for contract-based design”. In: *Proc. of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, pp. 21–28.
- Cimatti, A. and S. Tonetta (2015). “Contracts-refinement proof system for component-based embedded systems”. *Science of computer programming* 97, pp. 333–348.
- Clarke, E. M. and E. A. Emerson (1981). “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Proc. of the Workshop on Logic of Programs (WLP)*. Springer, pp. 52–71.
- Clarke, E. M., T. A. Henzinger, H. Veith, and R. Bloem (2018). *Handbook of model checking*. Vol. 10. Springer.
- Cohen, J. (2013). *Statistical power analysis for the behavioral sciences*. Academic press.
- Cok, D. R. (2011). “OpenJML: JML for Java 7 by extending OpenJDK”. In: *Proc. of the NASA Formal Methods Symposium (NFM)*. Springer, pp. 472–479.
- Cok, D. R. (2018). “Java automated deductive verification in practice: lessons from industrial proof-based projects”. In: *Proc. of the Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Springer, pp. 176–193.
- Coq. Coq Development Team. *The Coq proof assistant*, 1989-2021. <http://coq.inria.fr>. Accessed: 2021-06-18.
- Cuijpers, P. J. L. and M. A. Reniers (2005). “Hybrid process algebra”. *The Journal of Logic and Algebraic Programming* 62.2, pp. 191–245.
- Cuoq, P., F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski (2012). “Frama-C”. In: *Proc. of the Intl. Conference on Software Engineering and Formal Methods (SEFM)*. Springer, pp. 233–247.
- Czarnecki, K. and U. W. Eisenecker (2000). *Generative programming*.
- Dantzig, G. B. (1998). *Linear programming and extensions*. Princeton university press.

- Daran, M. and P. Thévenod-Fosse (1996). “Software error analysis: A real case study involving real faults and mutations”. *ACM SIGSOFT Software Engineering Notes* 21.3, pp. 158–171.
- Darke, P., S. Prabhu, B. Chimdyalwar, A. Chauhan, S. Kumar, A. Basakchowdhury, R. Venkatesh, A. Datar, and R. K. Medicherla (2018). “VeriAbs: Verification by abstraction and test generation”. In: *Proc. of the Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, pp. 457–462.
- De Alfaro, L. and T. A. Henzinger (2005). “Interface-based design”. In: *Engineering theories of software intensive systems*. Springer, pp. 83–104.
- De Gouw, S., F. S. de Boer, R. Bubel, R. Hähnle, J. Rot, and D. Steinhöfel (2019). “Verifying OpenJDKs sort method for generic collections”. *Journal of Automated Reasoning* 62.1, pp. 93–126.
- De Gouw, S., J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle (2015). “OpenJDKs Java.utils.Collection.sort() is broken: The good, the bad and the worst case”. In: *Proc. of the Intl. Conference on Computer Aided Verification (CAV)*. Springer, pp. 273–289.
- De Moura, L. and N. Björner (2008). “Z3: An efficient SMT solver”. In: *Proc. of the Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, pp. 337–340.
- De Win, B., F. Piessens, W. Joosen, and T. Verhanneman (2002). “On the importance of the separation-of-concerns principle in secure software engineering”. In: *Proc. of the Workshop on the Application of Engineering Principles to System Security Design*. Citeseer, pp. 1–10.
- DeMillo, R. A., R. J. Lipton, and F. G. Sayward (1978). “Hints on test data selection: Help for the practicing programmer”. *Computer* 11.4, pp. 34–41.
- Dijkstra, E. W. (1975). “Guarded commands, nondeterminacy and formal derivation of programs”. *Communications of the ACM* 18.8, pp. 453–457.
- Dijkstra, E. W. (1972). “Notes on structured programming”. In: *Structured Programming*. Academic Press Inc., pp. 1–82.
- Dijkstra, E. W. (1976). *A discipline of programming*. Prentice Hall PTR.
- Doyen, L., G. Frehse, G. J. Pappas, and A. Platzer (2018). “Verification of hybrid systems”. In: *Handbook of Model Checking*. Springer, pp. 1047–1110.
- Drozдов, D., S. Patil, V. Dubinin, and V. Vyatkin (2019). “Towards formal ASM semantics of timed control systems for industrial CPS”. In: *Proc. of the Intl. Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, pp. 1682–1685.
- Du Bousquet, L. and M. Lévy (2010). “Proof process evaluation with mutation analysis”. In: *Proc. of the Intl. Conference on Tests and Proof (TAP)*. Springer, pp. 55–60.
- Duggirala, P. S., S. Mitra, M. Viswanathan, and M. Potok (2015). “C2E2: A verification tool for stateflow models”. In: *Proc. of the Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, pp. 68–82.

- Elmqvist, H., S. E. Mattsson, and M. Otter (2001). "Object-oriented and hybrid modeling in modelica". *Journal Européen des systèmes automatisés* 35.4, pp. 395–404.
- Engel, C., A. Roth, A. Blome, R. Bubel, and S. Greiner (2010). *KeY quicktour for JML*. URL: <https://illwww.iti.kit.edu/~projekt/download/quicktour/quicktour-1.6.pdf>.
- Ernst, M. D., J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao (2007). "The Daikon system for dynamic detection of likely invariants". *Science of Computer Programming* 69.1-3, pp. 35–45.
- Feiler, P. H. and D. P. Gluch (2012). *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley.
- Filieri, A., H. Hoffmann, and M. Maggio (2015). "Automated multi-objective control for self-adaptive software design". In: *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*. ACM, pp. 13–24.
- Floyd, R. W. (1967). "Nondeterministic algorithms". *Journal of the ACM (JACM)* 14.4, pp. 636–644.
- Forsberg, K. and H. Mooz (1991). "The relationship of system engineering to the project cycle". In: *Proc. of the International Symposium (INCOSE)*. Vol. 1. 1. Wiley Online Library, pp. 57–65.
- France, R., A. Evans, K. Lano, and B. Rumpe (1998). "The UML as a formal modeling notation". *Computer Standards & Interfaces* 19.7, pp. 325–334.
- Frehse, G., Z. Han, and B. Krogh (2004). "Assume-guarantee reasoning for hybrid I/O-automata by over-approximation of continuous interaction". In: *Proc. of the Conference on Decision and Control (CDC)*. Vol. 1. IEEE, pp. 479–484.
- Frehse, G., C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler (2011). "SpaceEx: Scalable verification of hybrid systems". In: *Proc. of the Intl. Conference on Computer Aided Verification (CAV)*. Springer, pp. 379–395.
- Friedenthal, S., A. Moore, and R. Steiner (2014). *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann.
- Fulton, N., S. Mitsch, B. Bohrer, and A. Platzer (2017). "Bellerophon: Tactical theorem proving for hybrid systems". In: *Proc. of the Intl. Conference on Interactive Theorem Proving (ITP)*. Springer, pp. 207–224.
- Fulton, N., S. Mitsch, J.-D. Quesel, M. Völpl, and A. Platzer (2015). "KeYmaera X: An axiomatic tactical theorem prover for hybrid systems". In: *Proc. of the Intl. Conference on Automated Deduction (CADE)*. Springer, pp. 527–538.
- Furia, C. A., C. M. Poskitt, and J. Tschannen (2015). "The AutoProof Verifier: Usability by Non-Experts and on Standard Code". In: *Proc. of the Intl. Workshop on Formal Integrated Development Environment (F-IDE)*. Vol. 187. Open Publishing Association, pp. 42–55.
- Garvin, B. J., M. B. Cohen, and M. B. Dwyer (2011). "Evaluating improvements to a meta-heuristic search for constrained interaction testing". *Empirical Software Engineering* 16.1, pp. 61–102.

- Gentzen, G. (1935a). “Untersuchungen über das logische Schliessen. I.” *Mathematische zeitschrift* 35.
- Gentzen, G. (1935b). “Untersuchungen über das logische Schliessen. II”. *Mathematische Zeitschrift* 39.1, pp. 405–431.
- Gentzen, G. (1964). “Investigations into logical deduction”. *American philosophical quarterly* 1.4, pp. 288–306.
- Gerhart, S. L. (1975). “Correctness-preserving program transformations”. In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*, pp. 54–66.
- Ghassabani, E., A. Gacek, and M. W. Whalen (2016). “Efficient generation of inductive validity cores for safety properties”. In: *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*, pp. 314–325.
- Ghassabani, E., A. Gacek, M. W. Whalen, M. P. Heimdahl, and L. Wagner (2017). “Proof-based coverage metrics for formal verification”. In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, pp. 194–199.
- Gleirscher, M., S. Foster, and J. Woodcock (2019). “New opportunities for integrated formal methods”. *ACM Computing Surveys (CSUR)* 52.6, pp. 1–36.
- Gleirscher, M. and D. Marmosoler (2018). “Formal methods: Oversold? Underused? A survey”. *Computing Research Repository (CoRR)* abs/1812.08815. arXiv: 1812.08815. URL: <http://arxiv.org/abs/1812.08815>.
- Goebel, R., R. G. Sanfelice, and A. R. Teel (2012). *Hybrid dynamical systems*. Princeton University Press.
- Gómez, F. J., M. A. Aguilera, S. H. Olsen, and L. Vanfretti (2020). “Software requirements for interoperable and standard-based power system modeling tools”. *Simulation Modelling Practice and Theory* 103, p. 102095.
- Gössler, G. and J. Sifakis (2005). “Composition for component-based modeling”. *Science of Computer Programming* 55.1-3, pp. 161–183.
- Goswami, D., R. Schneider, A. Masrur, M. Lukaszewycz, S. Chakraborty, H. Voit, and A. Annaswamy (2012). “Challenges in Automotive Cyber-Physical Systems Design”. In: *Proc. of the Intl. Conference on Embedded Computer Systems (SAMOS)*, pp. 346–354. DOI: 10.1109/SAMOS.2012.6404199.
- Grebhahn, A., N. Siegmund, S. Apel, S. Kuckuk, C. Schmitt, and H. Köstler (2014). “Optimizing performance of stencil code with SPL conqueror”. In: *Proc. of the Intl. Workshop on High-Performance Stencil Computations (HiStencils)*, pp. 7–14.
- Grebing, S. and M. Ulbrich (2020). “Usability Recommendations for User Guidance in Deductive Program Verification”. In: *Deductive Software Verification: Future Perspectives*. Springer, pp. 261–284.
- Groce, A., I. Ahmed, C. Jensen, P. E. McKenney, and J. Holmes (2018). “How verified (or tested) is my code? Falsification-driven verification and testing”. *Automated Software Engineering* 25.4, pp. 917–960.



- Grün, B. J., D. Schuler, and A. Zeller (2009). “The impact of equivalent mutants”. In: *Proc. of the Intl. Conference on Software Testing, Verification, and Validation Workshops (ICST)*. IEEE, pp. 192–199.
- Guo, J., K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski (2013). “Variability-aware performance prediction: A statistical learning approach”. In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, pp. 301–311.
- Guo, J., D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu (2018). “Data-efficient performance learning for configurable systems”. *Empirical Software Engineering* 23.3, pp. 1826–1867.
- Ha, H. and H. Zhang (2019). “DeepPerf: performance prediction for configurable software with deep sparse neural network”. In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, pp. 1095–1106.
- Haber, A. (2016). *MontiArc – Architectural Modeling and Simulation of Interactive Distributed Systems*. Vol. 24. Shaker Verlag GmbH.
- Hähnle, R. and M. Huisman (2017). “24 Challenges in Deductive Software Verification.” In: *Proc. of the Intl. Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements (ARCADE)*, pp. 37–41.
- Hähnle, R. and M. Huisman (2019). “Deductive software verification: from pen-and-paper proofs to industrial tools”. In: *Computing and Software Science*. Springer, pp. 345–373.
- Hähnle, R., I. Schaefer, and R. Bubel (2013). “Reuse in software verification by abstract method calls”. In: *Proc. of the Intl. Conference on Automated Deduction (CADE)*. Springer, pp. 300–314.
- Al-Hajjaji, M., S. Krieter, T. Thüm, M. Lochau, and G. Saake (2016). “IncLing: efficient product-line testing using incremental pairwise sampling”. In: *Proc. of the Intl. Conference on Generative Programming: Concepts and Experiences (GPCE)*. Vol. 52. 3. ACM, pp. 144–155.
- Harel, D. (1987). “Statecharts: A visual formalism for complex systems”. *Science of Computer Programming* 8.3, pp. 231–274.
- Harel, D. and M. Politi (1998). *Modeling reactive systems with statecharts: the STATEMATE approach*. McGraw-Hill, Inc.
- Harris, M. (2016). “Google reports self-driving car mistakes: 272 failures and 13 near misses”. *The Guardian*. URL: <https://www.theguardian.com/technology/2016/jan/12/google-self-driving-cars-mistakes-data-reports>.
- Hatcliff, J., G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson (2012). “Behavioral interface specification languages”. *ACM Computing Surveys (CSUR)* 44.3, pp. 1–58.
- Havelund, K. and T. Pressburger (2000). “Model checking Java programs using Java Pathfinder”. *International Journal on Software Tools for Technology Transfer* 2.4, pp. 366–381.



- Helm, D., F. Kübler, M. Eichberg, M. Reif, and M. Mezini (2018). “A unified lattice model and framework for purity analyses”. In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, pp. 340–350.
- Henard, C., M. Papadakis, M. Harman, and Y. Le Traon (2015). “Combining multi-objective search and constraint solving for configuring large software product lines”. In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE Press, pp. 517–528.
- Henzinger, T. A. (2000). “The theory of hybrid automata”. In: *Verification of digital and hybrid systems*. Springer, pp. 265–292.
- Henzinger, T. A., M. Minea, and V. Prabhu (2001). “Assume-guarantee reasoning for hierarchical hybrid systems”. In: *Proc. of the Intl. Workshop on Hybrid Systems: Computation and Control (HSCC)*. Springer, pp. 275–290.
- Henzinger, T. A. and J. Sifakis (2007). “The Discipline of Embedded Systems Design”. *Computer* 40.10, pp. 32–40.
- Hiep, H.-D. A., J. Bian, F. S. de Boer, and S. de Gouw (2020). “A Tutorial on Verifying LinkedList Using KeY”. *Deductive Software Verification: Future Perspectives*, pp. 221–245.
- Hilbert, D. and W. Ackermann (1999). *Principles of mathematical logic*. Vol. 69. American Mathematical Soc.
- Hoare, C. (1972). “Proof of correctness of data representations”. *Acta Informatica* 1.4, pp. 271–281.
- Hoare, C. A. R. (1969). “An axiomatic basis for computer programming”. *Communications of the ACM* 12.10, pp. 576–580. DOI: <https://doi.org/10.1145/363235.363259>.
- Hoare, C. A. R. (Feb. 1981). “The Emperor’s Old Clothes”. *Commun. ACM* 24.2, pp. 75–83. DOI: [10.1145/358549.358561](https://doi.org/10.1145/358549.358561). URL: <https://doi.org/10.1145/358549.358561>.
- Holzmann, G. J. (1997). “The model checker SPIN”. *IEEE Transactions on Software Engineering (TSE)* 23.5, pp. 279–295.
- Hou, S.-S., L. Zhang, T. Xie, H. Mei, and J.-S. Sun (2007). “Applying interface-contract mutation in regression testing of component-based software”. In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, pp. 174–183.
- IEC 61508 (2011). *IEC: 61508 – Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. Accessed: 2021-18-06. URL: <https://www.vde-verlag.de/iec-normen/217177/iec-61508-1-2010.html>.
- ISO 21448 (2011). *ISO: 21448 – Road vehicles - Safety of the intended functionality*. Accessed: 2021-12-07. URL: <https://www.iso.org/standard/70939.html>.
- ISO 26262 (2011). *ISO: 26262 – Road vehicles - Functional safety*. Accessed: 2021-18-06. URL: <https://www.iso.org/standard/43464.html>.

- Jackson, D. (2002). “Alloy: a lightweight object modelling notation”. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.2, pp. 256–290.
- Jamshidi, P. and G. Casale (2016). “An uncertainty-aware approach to optimal configuration of stream processing systems”. In: *Proc. of the Intl. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, pp. 39–48.
- Jang, D., Z. Tatlock, and S. Lerner (2012). “Establishing browser security guarantees through formal shim verification”. In: *Proc. of the Intl. USENIX Security Symposium (USENIX Security)*, pp. 113–128.
- Jeannin, J.-B., K. Ghorbal, Y. Kouskoulas, A. Schmidt, R. Gardner, S. Mitsch, and A. Platzer (2017). “A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system”. *International Journal on Software Tools for Technology Transfer* 19.6, pp. 717–741.
- Jia, Y. and M. Harman (2010). “An analysis and survey of the development of mutation testing”. *IEEE transactions on Software Engineering* 37.5, pp. 649–678.
- Johansen, M. F., . Haugen, and F. Fleurey (2011). “Properties of realistic feature models make combinatorial testing of product lines feasible”. In: *Proc. of the Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, pp. 638–652.
- Johansen, M. F., . Haugen, and F. Fleurey (2012). “An algorithm for generating t-wise covering arrays from large feature models”. In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. ACM, pp. 46–55.
- Johnsen, E. B., R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen (2010). “ABS: A Core Language for Abstract Behavioral Specification”. In: *Proc. of the Intl. Symposium on Formal Methods for Components and Objects (FMCO)*. Springer, pp. 142–164.
- Jue, W., Y. Song, X. Wu, and W. Dai (2019). “A semi-formal requirement modeling pattern for designing industrial cyber-physical systems”. In: *Proc. of the Annual Conference of the IEEE Industrial Electronics Societ (IES)*. Vol. 1. IEEE, pp. 2883–2888.
- Just, R., D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser (2014). “Are mutants a valid substitute for real faults in software testing?” In: *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*, pp. 654–665.
- Kale, M. (2021). “Vollständige und Korrekte Implementierung von Skill Graphen”. BA thesis. Technical University of Braunschweig.
- Kaltenecker, C., A. Grebhahn, N. Siegmund, J. Guo, and S. Apel (2019). “Distance-based sampling of software configuration spaces”. In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, pp. 1084–1094.
- Kamburjan, E., S. Mitsch, and R. Hähnle (2022). “A Hybrid Programming Language for Formal Modeling and Verification of Hybrid Systems”. *Leibniz Transactions on Embedded Systems (LITES)*.

- Kang, E., S. Adep, D. Jackson, and A. P. Mathur (2016). “Model-based security analysis of a water treatment system”. In: *Proc. of the Intl. Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*. IEEE, pp. 22–28.
- Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson (1990). *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
- Khomh, F., B. Adams, T. Dhaliwal, and Y. Zou (2015). “Understanding the Impact of Rapid Releases on Software Quality”. *Empirical Software Engineering* 20.2, pp. 336–373.
- Kienzle, J., G. Mussbacher, P. Collet, and O. Alam (2016). “Delaying decisions in variable concern hierarchies”. In: *Proc. of the Intl. Conference on Generative Programming: Concepts and Experiences (GPCE)*. Vol. 52. 3. ACM, pp. 93–103.
- Kintis, M., M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman (2017). “Detecting trivial mutant equivalences via compiler optimisations”. *IEEE Transactions on Software Engineering* 44.4, pp. 308–333.
- Klein, G., J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser (2014). “Comprehensive formal verification of an OS microkernel”. *ACM Transactions on Computer Systems (TOCS)* 32.1, pp. 1–70.
- Klein, G., K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. (2009). “seL4: Formal verification of an OS kernel”. In: *Proc. of the Intl. Symposium on Operating Systems Principles (SOSP)*, pp. 207–220.
- Knüppel, A., I. Jatzkowski, M. Nolte, T. Thüm, T. Runge, and I. Schaefer (2020a). “Skill-Based Verification of Cyber-Physical Systems”. In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Ed. by H. Wehrheim and J. Cabot. Vol. 12076. Lecture Notes in Computer Science. Springer, pp. 203–223. DOI: [10.1007/978-3-030-45234-6\\_10](https://doi.org/10.1007/978-3-030-45234-6_10). URL: [https://doi.org/10.1007/978-3-030-45234-6\\_10](https://doi.org/10.1007/978-3-030-45234-6_10).
- Knüppel, A., T. Runge, and I. Schaefer (2020b). “Scaling Correctness-by-Construction”. In: *Proc. of the Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Ed. by T. Margaria and B. Steffen. Vol. 12476. Lecture Notes in Computer Science. Springer, pp. 187–207. DOI: [10.1007/978-3-030-61362-4\\_10](https://doi.org/10.1007/978-3-030-61362-4_10). URL: [https://doi.org/10.1007/978-3-030-61362-4\\_10](https://doi.org/10.1007/978-3-030-61362-4_10).
- Knüppel, A., L. Schaer, and I. Schaefer (2021a). “How much Specification is Enough? Mutation Analysis for Software Contracts”. In: *Proc. in the Intl. Conference on Formal Methods in Software Engineering (FormalISE)*. Ed. by S. Bliudze, S. Gnesi, N. Plat, and L. Semini. IEEE, pp. 42–53. DOI: [10.1109/FormalISE52586.2021.00011](https://doi.org/10.1109/FormalISE52586.2021.00011). URL: <https://doi.org/10.1109/FormalISE52586.2021.00011>.
- Knüppel, A., T. Thüm, C. Pardylla, and I. Schaefer (2018a). “Experience Report on Formally Verifying Parts of OpenJDK’s API with KeY”. In: *Proc. of the Intl. Workshop on Formal Integrated Development*

- Environment (F-IDE)*. Ed. by P. Masci, R. Monahan, and V. Prevosto. Vol. 284. EPTCS, pp. 53–70. DOI: [10.4204/EPTCS.284.5](https://doi.org/10.4204/EPTCS.284.5). URL: <https://doi.org/10.4204/EPTCS.284.5>.
- Knüppel, A., T. Thüm, C. Pardylla, and I. Schaefer (2018b). “Scalability of Deductive Verification Depends on Method Call Treatment”. In: *Proc. of the Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Ed. by T. Margaria and B. Steffen. Vol. 11247. Lecture Notes in Computer Science. Springer, pp. 159–175. DOI: [10.1007/978-3-030-03427-6\\_15](https://doi.org/10.1007/978-3-030-03427-6_15). URL: [https://doi.org/10.1007/978-3-030-03427-6%5C\\_15](https://doi.org/10.1007/978-3-030-03427-6%5C_15).
- Knüppel, A., T. Thüm, C. I. Pardylla, and I. Schaefer (2018c). “Understanding Parameters of Deductive Verification: An Empirical Investigation of KeY”. In: *Proc. of the Intl. Conference on Interactive Theorem Proving (ITP)*. Ed. by J. Avigad and A. Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Springer, pp. 342–361. DOI: [10.1007/978-3-319-94821-8\\_20](https://doi.org/10.1007/978-3-319-94821-8_20). URL: [https://doi.org/10.1007/978-3-319-94821-8%5C\\_20](https://doi.org/10.1007/978-3-319-94821-8%5C_20).
- Knüppel, A., T. Thüm, and I. Schaefer (2021b). “GUIDO: Automated Guidance for the Configuration of Deductive Program Verifiers”. In: *Proc. in the Intl. Conference on Formal Methods in Software Engineering (FormalISE)*. Ed. by S. Bliudze, S. Gnesi, N. Plat, and L. Semini. IEEE, pp. 124–129. DOI: [10.1109/FormalISE52586.2021.00018](https://doi.org/10.1109/FormalISE52586.2021.00018). URL: <https://doi.org/10.1109/FormalISE52586.2021.00018>.
- Koenig, N. and A. Howard (2004). “Design and use paradigms for Gazebo, an open-source multi-robot simulator”. In: *Proc. of the Intl. Conference on Intelligent Robots and Systems (IROS)*. Vol. 3. IEEE, pp. 2149–2154.
- Koopman, P. and M. Wagner (2016). “Challenges in autonomous vehicle testing and validation”. *SAE International Journal of Transportation Safety* 4.1, pp. 15–24.
- Kordon, F., J. Hugues, and X. Renault (2008). “From Model Driven Engineering to Verification Driven Engineering”. In: *Proc. of the Intl. Workshop on Software Technologies for Embedded and Ubiquitous Systems*. Ed. by U. Brinkschulte, T. Givargis, and S. Russo. Berlin, Heidelberg: Springer, pp. 381–393. DOI: [10.1007/978-3-540-87785-1\\_34](https://doi.org/10.1007/978-3-540-87785-1_34).
- Koubâa, A. et al. (2017). *Robot Operating System (ROS)*. Vol. 1. Springer.
- Kourie, D. G. and B. W. Watson (2012). *The Correctness-by-Construction Approach to Programming*. Springer Science & Business Media.
- Kovács, L. and A. Voronkov (2013). “First-order theorem proving and Vampire”. In: *Proc. of the Intl. Conference on Computer Aided Verification (CAV)*. Springer, pp. 1–35.
- Kroening, D. and M. Tautschnig (2014). “CBMC–C bounded model checker”. In: *Proc. of the Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, pp. 389–391.
- Kumar, R., M. O. Myreen, M. Norrish, and S. Owens (2014). “CakeML: a verified implementation of ML”. *ACM SIGPLAN Notices* 49.1, pp. 179–191.

- Kutluay, E. (2013). *Development and demonstration of a validation methodology for vehicle lateral dynamics simulation models*. VDI-Verlag Düsseldorf, Germany.
- Laibinis, L., A. Iliasov, and A. Romanovsky (2021). “Mutation Testing for Rule-Based Verification of Railway Signaling Data”. *IEEE Transactions on Reliability* 70.2, pp. 676–691.
- Lau, K.-K. and C. M. Tran (2012). “X-MAN: An MDE Tool for Component-Based System Development”. In: *Proc. of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, pp. 158–165.
- Le Guernic, C. (2009). “Reachability analysis of hybrid systems with linear continuous dynamics”. PhD thesis. Université Joseph-Fourier-Grenoble I.
- Le Traon, Y., B. Baudry, and J.-M. Jézéquel (2006). “Design by contract to improve software vigilance”. *IEEE Transactions on Software Engineering* 32.8, pp. 571–586.
- Leavens, G. T., A. L. Baker, and C. Ruby (2006). “Preliminary design of JML: A behavioral interface specification language for Java”. *ACM SIGSOFT Software Engineering Notes* 31.3, pp. 1–38. DOI: <https://doi.org/10.1145/1127878.1127884>.
- Leavens, G. T. and P. Muller (2007). “Information Hiding and Visibility in Interface Specifications”. In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, pp. 385–395.
- Lee, I., O. Sokolsky, S. Chen, J. Hatcliff, E. Jee, B. Kim, A. King, M. Mullen-Fortino, S. Park, A. Roederer, and K. K. Venkatasubramanian (2012). “Challenges and Research Directions in Medical CyberPhysical Systems”. *Proceedings of the IEEE* 100.1, pp. 75–90. DOI: [10.1109/JPROC.2011.2165270](https://doi.org/10.1109/JPROC.2011.2165270).
- Leino, K. R. M. (1998). “Data groups: Specifying the modification of extended state”. In: *Proc. of the Intl. Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 144–153.
- Leroy, X. (2009). “Formal verification of a realistic compiler”. *Communications of the ACM* 52.7, pp. 107–115.
- Liberzon, D. and A. S. Morse (1999). “Basic problems in stability and design of switched systems”. *IEEE control systems magazine* 19.5, pp. 59–70.
- Lin, Q., S. Adepu, S. Verwer, and A. Mathur (2018). “TABOR: A graphical model-based approach for anomaly detection in industrial control systems”. In: *Proc. of the Asia Conference on Computer and Communications Security (ASIACCS)*, pp. 525–536.
- Liskov, B. H. and J. Guttag (1986). *Abstraction and specification in program development*. Vol. 180. MIT press Cambridge.
- Liskov, B. H. and J. M. Wing (1994). “A behavioral notion of subtyping”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6, pp. 1811–1841.
- Liu, J., J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou (2010). “A calculus for hybrid CSP”. In: *Proc. of the Asian Symposium on Programming Languages and Systems*. Springer, pp. 1–15.

- Loos, S. M. and A. Platzer (2016). “Differential refinement logic”. In: *Proc. of the Symposium on Logic in Computer Science (LICS)*, pp. 505–514.
- Loos, S. M., A. Platzer, and L. Nistor (2011). “Adaptive cruise control: Hybrid, distributed, and now formally verified”. In: *Proc. of the International Symposium on Formal Methods (FM)*. Springer, pp. 42–56.
- Lynch, N., R. Segala, and F. Vaandrager (2003). “Hybrid I/O automata”. *Information and Computation* 185.1, pp. 105–157.
- Ma, Y.-S., J. Offutt, and Y.-R. Kwon (2006). “MuJava: a mutation system for Java”. In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*, pp. 827–830.
- Madeyski, L., W. Orzeszyna, R. Torkar, and M. Jozala (2013). “Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation”. *IEEE Transactions on Software Engineering* 40.1, pp. 23–42.
- Maler, O. and D. Nickovic (2004). “Monitoring temporal properties of continuous signals”. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, pp. 152–166.
- Mancini, T., F. Mari, I. Melatti, I. Salvo, E. Tronci, J. K. Gruber, B. Hayes, M. Prodanovic, and L. Elmegaard (2018). “Parallel statistical model checking for safety verification in smart grids”. In: *Proc. of the Intl. Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE, pp. 1–6.
- Maraninchi, F. and Y. Rémond (1998). “Mode-automata: About modes and states for reactive systems”. In: *Proc. of the European Symposium On Programming (ESOP)*. Springer, pp. 185–199.
- Maraninchi, F. and Y. Rémond (2003). “Mode-automata: a new domain-specific construct for the development of safe critical systems”. *Science of Computer Programming* 46.3, pp. 219–254.
- Massow, K. and I. Radusch (2018). “A rapid prototyping environment for cooperative advanced driver assistance systems”. *Journal of Advanced Transportation* 2018.
- Massow, K., F. Thiele, K. Schrab, S. Bunk, I. Tschinibaew, and I. Radusch (2020). “Scenario Definition for Prototyping Cooperative Advanced Driver Assistance Systems”. In: *Proc. of the Intl. Conference on Intelligent Transportation Systems (ITSC)*, pp. 1–8. DOI: [10.1109/ITSC45102.2020.9294238](https://doi.org/10.1109/ITSC45102.2020.9294238).
- Mathur, A. P. (2013). *Foundations of software testing, 2/e*. Pearson Education India.
- Mathur, A. P. and W. E. Wong (1994). “An empirical comparison of data flow and mutation-based test adequacy criteria”. *Software Testing, Verification and Reliability* 4.1, pp. 9–31.
- Maurer, M. (2000). “Flexible Automatisierung von Strassenfahrzeugen mit Rechnersehen”. PhD thesis. VDI-Verlag.
- McKillup, S. (2011). *Statistics Explained: An Introductory Guide for Life Scientists*. Cambridge University Press.



- McNemar, Q. (1947). "Note on the sampling error of the difference between correlated proportions or percentages". *Psychometrika* 12.2, pp. 153–157.
- Meinicke, J., T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake (2017). *Mastering software variability with FeatureIDE*. Springer.
- Meinicke, J., T. Thüm, R. Schröter, S. Krieter, F. Benduhn, G. Saake, and T. Leich (2016). "FeatureIDE: taming the preprocessor wilderness". In: *Proc. of the Intl. Conference on Software Engineering Companion (ICSE-Companion)*. ACM, pp. 629–632.
- Mendonca, M., A. Wasowski, and K. Czarnecki (2009). "SAT-based analysis of feature models is easy". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*, pp. 231–240.
- Mesarovic, M. D., D. Macko, and Y. Takahara (2000). *Theory of hierarchical, multilevel, systems*. Elsevier.
- Metsälä, S., K. Gulzar, V. Vyatkin, L. Gröhn, E. Väänänen, L. Saikko, and M. Nyholm (2017). "Simulation-enhanced development of industrial cyber-physical systems using OPC-UA and IEC 61499". In: *Proc. of the Intl. Conference on Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS)*. Springer, pp. 125–139.
- Meyer, B. (1988). "Eiffel: A language and environment for software engineering". *Journal of Systems and Software* 8.3, pp. 199–246.
- Meyer, B. (1992). "Applying 'design by contract'". *Computer* 25.10, pp. 40–51.
- Michniewicz, J. and G. Reinhart (2014). "Cyber-physical Robotics Automated Analysis, Programming and Configuration of Robot Cells based on Cyber-physical-systems". *Procedia Technology* 15. 2nd International Conference on System-Integrated Intelligence: Challenges for Product and Production Engineering, pp. 566–575. DOI: [10.1016/j.protcy.2014.09.017](https://doi.org/10.1016/j.protcy.2014.09.017).
- Milanez, A., B. Lima, J. Ferreira, and T. Massoni (2017). "Nonconformance between programs and contracts: a study on C#/code contracts open source systems". In: *Proc. of the ACM Symposium on Applied Computing (SAC)*, pp. 1219–1224.
- Milanez, A., D. Sousa, T. Massoni, and R. Gheyi (2014). "JMLOK2: A tool for detecting and categorizing nonconformances". In: *Proc. of the The Brazilian Conference on Software: Practice and Theory (CBSoft)*, pp. 69–76.
- Milanez, A. F., T. L. Massoni, R. Gheyi, and C. Grande-PB-Brazil (2013). "Categorizing nonconformances between programs and their specifications". In: *Proc. of the Brazilian Workshop on Systematic and Automated Software Testing (CBSOFT/SAST)*.
- Misson, H. A., F. S. Gonçalves, and L. B. Becker (2019). "Applying integrated formal methods on CPS design". In: *Proc. of the Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE, pp. 1–8.
- Mitra, S. (2021). *Verifying Cyber-Physical Systems: A Path to Safe Autonomy*. MIT Press. 312 pp. URL: <https://mitpress.mit.edu/contributors/sayan-mitra>.
- Mitsch, S. and A. Platzer (2016). "ModelPlex: Verified runtime validation of verified cyber-physical system models". *Formal Methods in System Design* 49.1, pp. 33–74.



- Morgan, C. (1994). *Programming from Specifications*. Prentice Hall,
- Mostowski, W. (2005). “Formalisation and verification of Java Card security properties in dynamic logic”. In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, pp. 357–371.
- Mostowski, W. (2007). “Fully verified Java Card API reference implementation”. In: *Proc. of the Intl. Verification Workshop (VERIFY)*. Vol. 259, pp. 136–151.
- Moura, L. de, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer (2015). “The Lean theorem prover”. In: *Proc. of the Intl. Conference on Automated Deduction (CADE)*. Springer, pp. 378–388.
- Müller, A., S. Mitsch, W. Retschitzegger, and W. Schwinger (2020). “Towards CPS Verification Engineering”. In: *Proc. of the Intl. Conference on Information Integration and Web-Based Applications & Services. iiWAS '20*. Chiang Mai, Thailand: Association for Computing Machinery, pp. 367–371. DOI: [10.1145/3428757.3429146](https://doi.org/10.1145/3428757.3429146). URL: <https://doi.org/10.1145/3428757.3429146>.
- Müller, A., S. Mitsch, W. Retschitzegger, W. Schwinger, and A. Platzer (2018a). “Tactical contract composition for hybrid system component verification”. *International Journal on Software Tools for Technology Transfer* 20.6, pp. 615–643.
- Müller, A., S. Mitsch, W. Schwinger, and A. Platzer (2018b). “A Component-Based Hybrid Systems Verification and Implementation Tool in KeYmaera X (Tool Demonstration)”. In: *Cyber Physical Systems. Model-Based Design*. Springer, pp. 91–110.
- Müller, P., M. Schwerhoff, and A. J. Summers (2016). “Viper: A verification infrastructure for permission-based reasoning”. In: *Proc. of the Intl. Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, pp. 41–62.
- Nair, V., T. Menzies, N. Siegmund, and S. Apel (2018). “Faster discovery of faster system configurations with spectral learning”. *Proc. of the Intl. Conference on Automated Software Engineering (ASE)* 25.2, pp. 247–277.
- Necula, G. C. (1997). “Proof-carrying code”. In: *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, pp. 106–119.
- Neghina, M., C.-B. Zamfirescu, and K. Pierce (2020). “Early-stage analysis of cyber-physical production systems through collaborative modelling”. *Software and Systems Modeling* 19.3, pp. 581–600.
- Negri, E., L. Fumagalli, and M. Macchi (2017). “A review of the roles of digital twin in CPS-based production systems”. *Procedia Manufacturing* 11, pp. 939–948.
- Neyman, J. (1977). “Frequentist probability and frequentist statistics”. *Synthese*, pp. 97–131.
- Nipkow, T., M. Eberl, and M. P. Haslbeck (2020). “Verified Textbook Algorithms”. In: *ATVA*. Springer, pp. 25–53.
- Nipkow, T., L. C. Paulson, and M. Wenzel (2002). *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

- Nolte, M., G. Bagschik, I. Jatzkowski, T. Stolte, A. Reschka, and M. Maurer (2017). "Towards a skill- and ability-based development process for self-aware automated road vehicles". In: *Proc. of the Intl. Conference on Intelligent Transportation Systems (ITSC)*. IEEE, pp. 1–6. DOI: <https://doi.org/10.1109/ITSC.2017.8317814>.
- Nuzzo, P., J. Li, A. L. Sangiovanni-Vincentelli, Y. Xi, and D. Li (2019). "Stochastic assume-guarantee contracts for cyber-physical system design". *ACM Transactions on Embedded Computing Systems (TECS)* 18.1, pp. 1–26.
- Nuzzo, P., M. Lora, Y. A. Feldman, and A. L. Sangiovanni-Vincentelli (2018). "CHASE: Contract-based requirement engineering for cyber-physical system design". In: *Proc. of the Design, Automation & Test in Europe Conference (DATE)*. IEEE, pp. 839–844.
- Nuzzo, P., A. L. Sangiovanni-Vincentelli, D. Bresolin, L. Geretti, and T. Villa (2015). "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems". *Proceedings of the IEEE* 103.11, pp. 2104–2132.
- O'Hearn, P. W. (2018). "Continuous reasoning: scaling the impact of formal methods". In: *Proc. of the Intl. Symposium on Logic in Computer Science (LICS)*. ACM, pp. 13–25.
- Offutt, A. J. and R. H. Untch (2001). "Mutation 2000: Uniting the orthogonal". *Mutation Testing for the new Century*, pp. 34–44.
- Offutt, A. J. and J. M. Voas (1996). "Subsumption of condition coverage techniques by mutation testing". *Department of Information and Software Systems Engineering, George Mason University, Tech. Rep. ISSE-TR-96-100*.
- Olaechea, R., S. Stewart, K. Czarnecki, and D. Rayside (2012). "Modelling and multi-objective optimization of quality attributes in variability-rich software". In: *Proc. of the Intl. Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages (NFPinDSML)*. ACM, p. 2.
- Oliveira, M., A. Cavalcanti, and J. Woodcock (2003). "ArcAngel: a Tactic Language for Refinement". *Formal Aspects of Computing* 15.1, pp. 28–47.
- Ozkaya, M. (2017). "Visual Specification and Analysis of Contract-Based Software Architectures". *Journal of Computer Science and Technology* 32.5, pp. 1025–1043.
- Ozkaya, M. and C. Kloukinas (2014). "Design-by-Contract for Reusable Components and Realizable Architectures". In: *Proc. of the Intl. Symposium on Component-Based Software Engineering (CBSE)*, pp. 129–138.
- Pace, D. K. (2004). "Modeling and simulation verification and validation challenges". *Johns Hopkins APL technical digest* 25.2, pp. 163–172.
- Pagliari, L., R. Mirandola, and C. Trubiani (2020). "Engineering cyber-physical systems through performance-based modelling and analysis: A case study experience report". *Journal of Software: Evolution and Process* 32.1, e2179.

- Papadakis, M., Y. Jia, M. Harman, and Y. Le Traon (2015). “Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique”. In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. Vol. 1. IEEE, pp. 936–946.
- Papadakis, M., M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman (2019). “Mutation testing advances: an analysis and survey”. In: *Advances in Computers*. Vol. 112. Elsevier, pp. 275–378.
- Parnas, D. L. (1972). “On the criteria to be used in decomposing systems into modules”. In: *Pioneers and Their Contributions to Software Engineering*. Springer, pp. 479–498.
- Pellkofer, M. (2003). “Verhaltensentscheidung für autonome Fahrzeuge mit Blickrichtungss-teuerung”. PhD thesis. Universität der Bundeswehr München, Universitätsbibliothek.
- Pierce, B. C. and C. Benjamin (2002). *Types and programming languages*. MIT press.
- Platzer, A. (2008). “Differential Dynamic Logic for Hybrid Systems”. *Journal of Automated Reasoning* 41.2, pp. 143–189. DOI: [10.1007/s10817-008-9103-8](https://doi.org/10.1007/s10817-008-9103-8). URL: <https://doi.org/10.1007/s10817-008-9103-8>.
- Platzer, A. (2010). *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics*. Springer. DOI: [10.1007/978-3-642-14509-4](https://doi.org/10.1007/978-3-642-14509-4). URL: <https://doi.org/10.1007/978-3-642-14509-4>.
- Platzer, A. (2012). “Logics of Dynamical Systems”. In: *Proc. of the Intl. Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, pp. 13–24. DOI: [10.1109/LICS.2012.13](https://doi.org/10.1109/LICS.2012.13). URL: <https://doi.org/10.1109/LICS.2012.13>.
- Platzer, A. (2017). “A complete uniform substitution calculus for differential dynamic logic”. *Journal of Automated Reasoning* 59.2, pp. 219–265.
- Platzer, A. (2018). *Logical foundations of cyber-physical systems*. Springer.
- Pnueli, A. (1977). “The temporal logic of programs”. In: *Proc. of the Symposium on Foundations of Computer Science (SFCS)*. IEEE, pp. 46–57.
- Pohl, K., G. Böckle, and F. J. van Der Linden (2005). *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.
- Poland, K., M. P. McKay, D. Bruce, and E. Becic (2018). “Fatal crash between a car operating with automated control systems and a tractor-semitrailer truck”. *Traffic Injury Prevention* 19.sup2, S153–S156.
- Polikarpova, N., I. Ciupa, and B. Meyer (2009). “A comparative study of programmer-written and automatically inferred contracts”. In: *Proc. of the Intl. Symposium on Software Testing and Analysis (ISSTA)*, pp. 93–104.
- Polikarpova, N., C. A. Furia, Y. Pei, Y. Wei, and B. Meyer (2013). “What good are strong specifications?” In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, pp. 262–271.

- Ptolemaeus, C. (2014). *System design, modeling, and simulation: using Ptolemy II*. Vol. 1. Ptolemy.org Berkeley.
- Queille, J.-P. and J. Sifakis (1982). "Specification and verification of concurrent systems in CESAR". In: *Proc. of the Intl. Symposium on programming*. Springer, pp. 337–351.
- Quigley, M., K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, et al. (2009). "ROS: an open-source Robot Operating System". In: *Proc. of the Workshop on Open Source Software*. Vol. 3. 3.2. Kobe, Japan, p. 5.
- Rademaker, A., C. Braga, and A. Sztajnberg (2005). "A Rewriting Semantics for a Software Architecture Description Language". *Electronic Notes in Theoretical Computer Science* 130, pp. 345–377.
- Rao, A. C., A. Raouf, G. Dhadyalla, and V. Pasupuleti (2017). "Mutation testing based evaluation of formal verification tools". In: *Proc. of the Intl. Conference on Dependable Systems and Their Applications (DSA)*. IEEE, pp. 1–7.
- Rawat, D. B., J. J. Rodrigues, and I. Stojmenovic (2015). *Cyber-Physical Systems: From Theory to Practice*. CRC Press.
- Reschka, A. (2017). "Fertigkeiten-und Fähigkeitengraphen als Grundlage des sicheren Betriebs von automatisierten Fahrzeugen im öffentlichen Strassenverkehr in städtischer Umgebung". PhD thesis.
- Reschka, A., G. Bagschik, S. Ulbrich, M. Nolte, and M. Maurer (2015). "Ability and skill graphs for system modeling, online monitoring, and decision support for vehicle guidance systems". In: *Proc. of the IEEE Intelligent Vehicles Symposium (IV)*. IEEE, pp. 933–939.
- Reussner, R. H., H. W. Schmidt, and I. H. Poernomo (2003). "Reliability Prediction for Component-Based Software Architectures". *Journal of Systems and Software* 66.3, pp. 241–252.
- Rice, D. (2019). "The Driverless Car and the Legal System: Hopes and Fears as the Courts, Regulatory Agencies, Waymo, Tesla, and Uber deal with this Exciting and Terrifying new Technology". *Journal of Strategic Innovation and Sustainability (JSIS)* 14.1, pp. 134–146.
- Richter, C. and H. Wehrheim (2019). "Pesco: Predicting sequential combinations of verifiers". In: *Proc. of the Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, pp. 229–233.
- Ringer, T., N. Yazdani, J. Leo, and D. Grossman (2018). "Adapting proof automation to adapt proofs". In: *Proc. of the Intl. Conference on Certified Programs and Proofs (CPP)*, pp. 115–129.
- Roy, C. and W. Oberkampff (2010). "A complete framework for verification, validation, and uncertainty quantification in scientific computing". In: *Proc. of the AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, p. 124.
- Rozier, K. Y. (2016). "Specification: The biggest bottleneck in formal methods and autonomy". In: *Proc. of the Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. Springer, pp. 8–26.

- Ruchkin, I., J. Sunshine, G. Iraci, B. Schmerl, and D. Garlan (2018). “IPL: an integration property language for multi-model cyber-physical systems”. In: *Proc. of the International Symposium on Formal Methods (FM)*. Springer, pp. 165–184.
- Runge, T., A. Knüppel, T. Thüm, and I. Schaefer (2020). “Lattice-Based Information Flow Control-by-Construction for Security-by-Design”. In: *Proc. in the Intl. Conference on Formal Methods in Software Engineering (FormaliSE)*. ACM, pp. 44–54. DOI: [10.1145/3372020.3391565](https://doi.org/10.1145/3372020.3391565). URL: <https://doi.org/10.1145/3372020.3391565>.
- Runge, T., I. Schaefer, L. Cleophas, T. Thüm, D. G. Kourie, and B. W. Watson (2019a). “Tool Support for Correctness-by-Construction”. In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Ed. by R. Hähnle and W. M. P. van der Aalst. Vol. 11424. Lecture Notes in Computer Science. Springer, pp. 25–42. DOI: [10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2). URL: [https://doi.org/10.1007/978-3-030-16722-6\\_2](https://doi.org/10.1007/978-3-030-16722-6_2).
- Runge, T., T. Thüm, L. Cleophas, I. Schaefer, and B. W. Watson (2019b). “Comparing correctness-by-construction with post-hoc verification: a qualitative user study”. In: *Proc. of the Intl. Symposium on Formal Methods – International Workshops (FM Workshops)*. Springer, pp. 388–405.
- Sampigethaya, K. and R. Poovendran (2013). “Aviation CyberPhysical Systems: Foundations for Future Aircraft and Air Transport”. *Proceedings of the IEEE* 101.8, pp. 1834–1855. DOI: [10.1109/JPROC.2012.2235131](https://doi.org/10.1109/JPROC.2012.2235131).
- Sangiovanni-Vincentelli, A., W. Damm, and R. Passerone (2012). “Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems”. *European Journal of Control* 18.3, pp. 217–238.
- Schmidt, D. C. (2006). “Model-driven engineering”. *IEEE Computer Society* 39.2, p. 25.
- Schumann, J. M. (2001). *Automated theorem proving in software engineering*. Springer Science & Business Media.
- Secleanu, C., M. Johansson, J. Suryadevara, G. Sapienza, T. Secleanu, S.-E. Ellevseth, and P. Pettersson (2017). “Analyzing a wind turbine system: From simulation to formal verification”. *Science of Computer Programming* 133, pp. 216–242.
- Shaffer, J. P. (1995). “Multiple hypothesis testing”. *Annual review of psychology* 46.1, pp. 561–584.
- Shah, S., D. Dey, C. Lovett, and A. Kapoor (2018). “Airsim: High-fidelity visual and physical simulation for autonomous vehicles”. In: *Proc. of the Intl. Conference on Field and service robotics (ICFSR)*. Springer, pp. 621–635.
- Siedersberger, K.-H. (2004). “Komponenten zur automatischen Fahrzeugführung in sehenden (semi-)autonomen Fahrzeugen”. PhD thesis. BU München.
- Siegmund, N., A. Grebhahn, S. Apel, and C. Kästner (2015). “Performance-influence models for highly configurable systems”. In: *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*, pp. 284–294.


- Siegmund, N., M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake (2012). “SPL Conqueror: Toward optimization of non-functional properties in software product lines”. *Software Quality Journal* 20.3-4, pp. 487–517.
- Stolte, T., A. Reschka, G. Bagschik, and M. Maurer (2015). “Towards Automated Driving: Unmanned Protective Vehicle for Highway Hard Shoulder Road Works”. In: *Proc. of the Intl. Conference on Intelligent Transportation Systems (ITSC)*. IEEE, pp. 672–677.
- Summers, A. J. and P. Müller (2018). “Automating deductive verification for weak-memory programs”. In: *Proc. of the Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, pp. 190–209.
- Tabuada, P. (2009). *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media. doi: 10.1007/978-1-4419-0224-5.
- Tan, Y. K. and A. Platzer (2021). “Switched systems as hybrid programs”. *IFAC-PapersOnLine* 54.5, pp. 247–252.
- Thüm, T., A. Knüppel, S. Krüger, S. Bolle, and I. Schaefer (2019). “Feature-oriented contract composition”. *Journal of Systems and Software* 152, pp. 83–107.
- Turing, A. M. (1937). “On computable numbers, with an application to the Entscheidungsproblem”. *Proc. of the London Mathematical Society* 2.1, pp. 230–265.
- UML 2 (2017). *OMG Unified Modeling Language Specification in Version 2.5.1*. <http://www.omg.org/spec/UML/2.5.1>. Accessed: 2021-10-04.
- Van Emden, E. and L. Moonen (2002). “Java quality assurance by detecting code smells”. In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, pp. 97–106.
- Varshosaz, M., M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer (2018). “A classification of product sampling for software product lines”. In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. Vol. 1. ACM, pp. 1–13.
- Wasilewska, A., Wasilewska, and Drougas (2018). *Logics for computer science*. Springer.
- Wei, Y., C. A. Furia, N. Kazmin, and B. Meyer (2011). “Inferring better contracts”. In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*, pp. 191–200.
- Weidenbach, C., D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischniewski (2009). “SPASS Version 3.5”. In: *Proc. of the Intl. Conference on Automated Deduction (CADE)*. Springer, pp. 140–145.
- Westman, J., M. Nyberg, and M. Törngren (2013). “Structuring safety requirements in ISO 26262 using contract theory”. In: *Proc. of the Intl. Conference on Computer Safety, Reliability, and Security (SAFECOMP)*. Springer, pp. 166–177.
- Winskel, G. (1993). *The formal semantics of programming languages: an introduction*. MIT press.
- Wirth, N. (2001). “Program development by stepwise refinement”. In: *Pioneers and Their Contributions to Software Engineering*. Springer, pp. 545–569.

- Wohlin, C., P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén (2012). *Experimentation in software engineering*. Springer Science & Business Media.
- Wu, F., W. Weimer, M. Harman, Y. Jia, and J. Krinke (2015). “Deep parameter optimisation”. In: *Proc. of the Intl. Genetic and Evolutionary Computation Conference (GECCO)*. ACM, pp. 1375–1382.
- Xi, B., Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang (2004). “A smart hill-climbing algorithm for application server configuration”. In: *Proc. of the Intl. Conference on World Wide Web*. ACM, pp. 287–296.
- Yigitbasi, N., T. L. Willke, G. Liao, and D. Epema (2013). “Towards machine learning-based auto-tuning of mapreduce”. In: *Proc. of the Intl. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, pp. 11–20.
- Yu, R., Y. Zhang, and J. Xuan (2020). “MetPurity: a learning-based tool of pure method identification for automatic test generation”. In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, pp. 1326–1330.
- Zhan, N., S. Wang, and H. Zhao (2013). “Formal modelling, analysis and verification of hybrid systems”. In: *Unifying Theories of Programming and Formal Engineering Methods*. Springer, pp. 207–281.
- Zhang, L. (2013). “Specifying and modeling automotive cyber physical systems”. In: *Proc. of the Intl. Conference on Computational Science and Engineering (CSE)*. IEEE, pp. 603–610.
- Zhang, L. (2014). “Modeling large scale complex cyber physical control systems based on system of systems engineering approach”. In: *Proc. of the Intl. Conference on Automation and Computing (ICAC)*. IEEE, pp. 55–60.
- Zhang, Y., J. Guo, E. Blais, and K. Czarnecki (2015). “Performance prediction of configurable software systems by Fourier learning (t)”. In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, pp. 365–373.
- Zheng, W., R. Bianchini, and T. D. Nguyen (2007). “Automatic configuration of internet services”. *ACM SIGOPS Operating Systems Review* 41.3, pp. 219–229.
- Zimmerman, D. W. (1998). “Invalidation of parametric and nonparametric statistical tests by concurrent violation of two assumptions”. *The Journal of experimental education* 67.1, pp. 55–68.









---

Technische Universität Carolo-Wilhelmina Braunschweig  
Institut für Softwaretechnik und Fahrzeuginformatik

Mühlenpfordtstr. 23  
D-38106 Braunschweig