



Solving the Bank

LIGHTWEIGHT SPECIFICATION AND VERIFICATION TECHNIQUES FOR ENTERPRISE SOFTWARE

JOUKE HARMEN STOEL

Solving the Bank

Lightweight Specification and Verification Techniques for
Enterprise Software

Solving the Bank

Lightweight Specification and Verification Techniques for Enterprise Software

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
aan de Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof. dr. S.K. Lenaerts, voor een
commissie aangewezen door het College voor
Promoties, in het openbaar te verdedigen
op 8 november 2023 om 16.00 uur

door

Jouke Harmen Stoel

geboren te Delfzijl

Promotiecommissie:

Voorzitter: prof. dr. M.A. Peletier

Promotores: prof. dr. J.J. Vinju (CWI - Technische Universiteit Eindhoven)
prof. dr. T. van der Storm (CWI - Rijksuniversiteit Groningen)

Copromotor: prof. dr. M.G.J. van den Brand

Overige leden: dr. E.M. Torlak (University of Washington)
prof. dr. M. Huisman (Universiteit Twente)
dr. M.A. Reniers
prof. dr. G.K. Keller (Universiteit Utrecht)
J. Bosman (ING Nederland NV)

Adviseur: J. Bosman (ING Nederland NV)

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



Centrum Wiskunde & Informatica

TU/e

ING 

axini



The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) under the auspices of the research school Institute for Programming research and Algorithmics (IPA) and has been supported by the ING.

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-5845-2.

Thesis cover art generated by Midjourney AI using the query “an iconic bank building made up of electronic circuits with a dollar sign logo on top in an isometric perspective. The bank building is standing on a large, colorful electronic motherboard. The background is white”.

This thesis is printed by ProefschriftMaken || www.proefschriftmaken.nl

CONTENTS

Contents	vii
Acknowledgments	xi
1 Introduction	1
1.1 Keeping Enterprise Software Systems Evolving	1
1.2 The Need for Explicit Knowledge	2
1.3 Formal Methods to Capture System Knowledge	2
1.4 Lightweight Formal Methods, a Pragmatic Middle ground	3
1.5 Research Context	4
1.6 Lightweight Formal Methods: a short and incomplete overview	5
1.7 Questions along the different axes of partiality	7
1.8 Origin of the Chapters	9
1.9 Software Artifacts	12
2 On the Design of the Rebel Specification Language and Its Application inside a Bank	15
2.1 Introduction	15
2.2 Background	16
2.3 Design of the Rebel Language and ISE	17
2.4 Rebel Specifications Explained	18
2.5 Simulation and Checking Specifications	21
2.6 Performing Model Based Testing of Existing Applications	26
2.7 Applying Rebel inside the Bank	27
2.8 Related Work	27
2.9 Conclusion	29
3 AlleAlle: Bounded Relational Model Finding with Unbounded Data	31
3.1 Introduction	31
3.2 ALLEALLE	33
3.3 Formal Definition of ALLEALLE	38
3.4 Translating ALLEALLE to SMT	42
3.5 Evaluation	53
3.6 Related Work	60
3.7 Conclusion	61
4 Constraint-based Run-time State Migration for Live Modeling	65

4.1	Introduction	65
4.2	Motivating Example	66
4.3	Structuring Constraints for Run-time State Migration	69
4.4	NEXTEP: a Language for State Migration	71
4.5	Evaluation	77
4.6	Related Work & Discussion	84
4.7	Conclusion & Future Work	87
5	Modeling with Mocking	91
5.1	Introduction	91
5.2	Rebel2 by Example: Money Transfer	93
5.3	Formalization	99
5.4	Implementation	102
5.5	Evaluation	108
5.6	Related Work	113
5.7	Conclusion	114
6	Design & Implementation	117
6.1	Rebel, version 1	117
6.2	AlleAlle	119
6.3	Nextep	131
6.4	REBEL2	139
6.5	Conclusion	144
7	Conclusion	147
7.1	Research Question 1: Partiality of Language	148
7.2	Research Question 2: Partiality of Modeling	149
7.3	Research Question 3: Partiality of Analysis	149
7.4	Research Question 4: Partiality of Composition	150
7.5	Future directions	150
7.6	Advice for our collaborating partner	152
	Bibliography	153
A	About Rascal	169
A.1	Rascal Programs	169
A.2	Rascal Syntax Definitions	169
B	Syntax definitions	171
B.1	Syntax Definition of REBEL	171
B.2	Syntax definition of ALLEALLE	175
B.3	Syntax definition of NEXTEP in Rascal	179

B.4	Syntax definition of REBEL2	181
C	Algorithms	185
C.1	AlleAlle algorithms	185
C.2	Key algorithms from the REBEL2 implementation	186
C.3	Applying 'Forget' and 'Mock'	191
C.4	Translation from Rebel2 to AlleAlle	192
D	Examples	221
D.1	Example translation from REBEL2 to ALLEALLE	221
E	Data	233
E.1	Optimal Package Dependency Resolution	233

ACKNOWLEDGMENTS

“And everything goes back to the beginning”*

It is strange to write this last and final chapter of this thesis. Looking back I have had the privilege to have met and worked with so many amazing people. That is the extra bonus for people that take their time ;) I have said it before but now I will also put it to writing: having done this PhD has been one of the best jobs I had so far and the people I have met are to thank for that!

First of all my gratitude goes out to my daily supervisors, Jurgen Vinju and Tijs van der Storm. Jurgen, I have never met another soul who is as positive as you. Even in troubled times you manage to find the silver lining and turn situations into learning experiences, for yourself and your surroundings. This is deeply inspiring and humbling. Next to that, your endless belief in my ability to finish this journey was of incredible support.

Tijs, thank you for your exceptional keen insights. You always seem to know how to break things down to their essence and remove all the unimportant bits. I greatly admire you for that and that made working together such a nice experience.

To the both of you I thank you for all your valuable lessons, companionship and support.

Mark, thank you for your guidance and taking me under your wings for 9 months. Teaching at the TU/e was a very insightful experience showing me how much fun it is to help others in their struggles to learn new things.

I would like to thank all the members of my committee, Joost Bosman Sr., Marieke Huisman, Gabriele Keller, Michel Reniers and Emina Torlak for their time and valuable feedback on the manuscript that lies before you.

An extra special thanks goes out to Joost Bosman Sr. of the ING, the instigator of all of this. Your belief and desire to further develop the state of practice of making software such that we are able to make stronger claims about its correctness, was inspiring and admirable. Thanks for giving me the opportunity to dive into this quest and give it my own personal twist.

Next to Joost Bosman Sr. there are many other people at the ING that I owe my gratitude. My thanks goes out to Robbert van Dalen, Jorryt Dijkstra, Luna Luo, Viet Nguyen, Rene Niekel, Alessandro Vermeulen, Kevin van der Vlist and Herbert van de Wetering. Special mention goes out to Herbert and Rene for helping out with putting my work to practice, Alessandro for being one of the early adopters of Rebel and to Viet who was brave enough to build a company on the ideas behind Rebel and code generation.

*from the song *Hollow Talk of the Choir of the Young Believers*.

I also look back with special fondness to my time at TU/e. In my (brief) time helping out as teacher I met with many kind and smart people of both the SET and FSA group. My thanks goes out to my office mates Yuexu (Celine) Chen, Josh Mengerink, Sander de Putter, Raquel Álvarez Ramirez and Arash Khabbaz Saberi, Next to my office mates there were many others at TU/e who I owe my thanks: Önder Babur, Olav Bunte, Loek Cleophas, Rick Erkens, Kees Huizing, Ruurd Kuiper, Maurice Laveaux, Thomas Neele, Agnes van Reek, Alexander Serebrenik, Mahmoud Talebi, Wessley Torres, Tom Verhoeff and many other colleagues. An extra mention goes out to Alexander for the enjoyable teaching we did together at the TU Twente.

The majority of my PhD I spent at CWI, in the SWAT group. An inspiring and tranquil place, a true safe haven for research. I wish to thank all my colleagues and friends that I have met over the years: Rodin Aarssen, Bas Basten, Nikolaos Bezirgiannis, Aggelos Biboudis, Thomas van Binsbergen, Jeroen van den Bos, Joost Bosman Jr., Yanja Dajsuren, Susanne van Dam, Thomas Degueule, Maarten Dijkema, Pablo Inostroza Valdera, Paul Klint, Davy Landman, Bert Lisser, Lina Ochea Venegas, Atze van der Ploeg, Riemer van Rozen, Tim Soethout, Michaël Steindorfer, Ulyana Tikhonova, Mauricio Verano Merino, Aiko Yamashita and many, many more. A special mention goes out to Tim for being such a nice collaborating partner and friend. We had many discussions about how you could go from a Rebel specification to an actual implementation. His work does precisely this. How cool is that!

During this last part of my journey I am employed by Axini. They provided me with the possibility to finish this thesis. Next to that, it is very cool to work at a place that builds a very usable product around a formal theory for Model Based Testing. It is nice to put theory into practice! Many thanks goes out to Tobias Bachmann, Mark Bebawy, Machiel van der Bijl, Bart-Jan Hilbrands, Menno Jonkers, Linda de Koter, Lars Meijer, Jorge Mora Perdiguero, Ruben Reijers, Serena Rietbergen, Theo Ruys, Lucas Steehouwer, Peter Verkade, Ivo Wever, Ronald Wilts and Floris Zeven.

During all this time I had the privilege to have many friends around me, keeping me sane. It is impossible to list them all here but I am grateful to each and every one of them. Special thanks goes out to Anne Seghers and Frank van Egmond for helping each other out during those crazy corona years. As the saying goes, it takes a village to raise a child. You guys were our village.

I also would like to express my gratitude to Sebastiaan Boekhoorn and Miron Nabokov for the many hours we spent making music and your good friendship. Thanks Jochem Noë for introducing me to bouldering and being such good company while going up the wall. Lastly, a special thanks to Tim van der Weerd, my good friend and paranymp, for all the good conversations and shared love for bad word puns.

This brings me to my family. I am very grateful to my parents Roelf and Heily Stoel for bringing me up the way they did. Thanks for all your love and guidance

and learning me the joy of creating and wondering. This is something that I feel is so essential for a lot of the things I have done in my grown-up live.

Thanks to my parent-in-laws, Cees and Carla Plat. You are always there for us when we need you. We are always welcomed with open arms while often having the most exquisit meals prepared for us!

Many thanks to my brother Gerhard for always supporting me no matter what choice I make. Having you, Marjet, Emke and Jitske so close by this last year has really been a joy.

Lastly, my biggest thanks and greatest gratitude goes out to my own family. When I started this PhD, my daughter Lucie was just born. Now she is 9 years old and already an amazing person of her own. Halfway the journey my just as amazing son Boris was born. He is also already finding his way at school and is learning to read, that is why I write the next part in Dutch for them:

Mijn allerliefste kinderen, ik ben zóóó ontzettend blij en dankbaar om jullie in mijn leven te mogen hebben. Misschien weten jullie het niet, maar jullie hebben het schrijven van 'dit boekje' zo veel fijner gemaakt. Jullie zorgen er altijd weer voor dat we vooral in het hier-en-nu moeten zijn, en dat is de meest belangrijke tijd om te zijn.

There can only be one to which I devote the last words of this chapter. Dear Sara, you don't know how grateful I am to have you by my side. By knowing that we do all these things together makes it so much easier and more enjoyable. Thanks for being there and sometimes nudging me to continue. Because of you I started studying again and the rest is history. I love you with all my heart.

Thank you all!

INTRODUCTION

Growing large Enterprise Software Systems like those found in banks is a daunting task. We use the word “growing” since large systems are never created in one go. They evolve. They are the result of endless iterations of building parts of the system. These different parts, often written by different people, in different time periods, using different techniques and languages, make up the system as a whole. Constructing and keeping these systems consistent and coherent is a challenging task.

1.1 KEEPING ENTERPRISE SOFTWARE SYSTEMS EVOLVING

In Lehman’s 1980 landmark paper on Software Evolution he describes five laws* which govern the evolution of real-world software systems [Leh80]. These laws describe why it is hard to maintain large, real-world systems.[†] They state that in order to keep a useful, functioning system it needs to be constantly adapted to meet the changing requirements of the organization (the Law of Continuous Change). These changes lead to an ever increasingly complex system (the Law of Increasing Complexity) and to keep the system evolving it is necessary that everyone that associates with it (users, developers, etc.) keeps a good understanding of its working (the law of Conservation of Organizational Stability and the Law of Conservation of Familiarity). If not, the evolution of the system comes to a halt and, ultimately, its demise since it cannot fulfill the law of Continuous Change anymore.

Enterprises struggle with the implications of these laws. Enterprise software has a tendency to exist for a long time [Fow02]. Keeping everyone that has a stake in these systems familiar with its working such that changes can still be made is challenging. This is even worsened when disruptive organizational changes happen such as reorganizations or mergers. Suddenly different, unfamiliar software systems must be integrated with existing systems, systems that were originally not meant to integrate with each other. This unfamiliarity (both in functionality and technology) makes assimilation of these systems in a coherent way hard, often resulting in large, expensive software projects of which its successful implementation is hard to predict [Cha05].

*In later work he extended these laws with the addition of three laws. [Leh96]

[†]Lehman introduced the classification of E-type systems for real-world systems that help with solving actual, real-world problems [Leh80].

1.2 THE NEED FOR EXPLICIT KNOWLEDGE

One would expect that the design and rules —i.e., business logic— of these systems would be thoroughly documented such that new stakeholders can get acquainted with the system. This is, however, often not the case. Most documentation, whether it is requirements, design or architectural, is only rarely, if ever, updated when the system is changed [LSF03]. The result of this is that newcomers can at best partly rely on the existing documentation but also need to consult other sources such as experience of others.

Ultimately it is the source code of the software itself that encapsulates the design and domain rules. However, recovering the needed details from source code is not an easy task and depends highly on a number of factors such as used software language and level of documentation in the code [Big89]. Source code is on a lower level of abstraction in which domain and technical concepts are weaved together and can be hard to unravel. As such, relying on the source code as the “source of truth” might be costly and inefficient.

1.3 FORMAL METHODS TO CAPTURE SYSTEM KNOWLEDGE

One way to capture system knowledge is make use of a *formal specification* methodology. Formal specifications allow to precisely capture (high- and low-level) system properties, both descriptive and prescriptive properties [Lam00]. According to Lamsweerde a formal specification is: “the expression in some formal language and at some level of abstraction, of a collection of properties that some system must satisfy” [Lam00]. The benefit of these formal and precise definitions of properties are that, unlike specifications in natural language, they do not leave any room for ambiguity. Next to that, most formal specification methods allow for some kind of animation or simulation, automatic verification or even sound refinement to source code from a specification. Examples of these methods are B [Abr96], VDM [BJ], ASM [GB95], mCRL₂ [GM14] and Promela [HL91]. The notion of using formal specification and reasoning techniques is by no means new. It originated in the early days of computer science (late 1940) and has seen a long line of research resulting in many different methodologies based on algebraic specifications, history based formalisms, state machines based and process algebras.

Although these formal methods are promising there are some often heard arguments that prohibit their use in industry scale projects:

- *Cost*: Applying formal methods to a large, industry scale project requires (at least initially) a significant high-level of expertise and time. Although there are success stories of the application of formal methods that show that making use of these methods increases the quality of the software while decreasing the overall development time [BBF⁺99], most often the use of these methods are

perceived as too time consuming and too hard to grasp resulting in the need of costly experts [RCB02].

- *Scope*: Using formal techniques for verification often reach the limits of our current state-of-the-art. For instance, a technique such as model checking is quickly plagued with the state-space explosion problem [CHV⁺18] or using automatic proof assistance often requires the user to still discharge many manual proof obligations.

A study amongst 216 industrial practitioners indicated that scalability, skills and education were the perceived key challenges with regards to the use of formal methods on industry scale systems [GM20]. These challenges probably contribute to the fact that there are not many known examples of large enterprises, outside the hardware domain (e.g. [AJM⁺00; KGN⁺09]) or safety critical software (i.e., NASA organizes a Formal Methods symposium each year), that make use of formal methods. Especially in a time where enterprises struggle with finding suitably skilled workers, finding those who are trained in the use of formal methods is nearly impossible.

1.4 LIGHTWEIGHT FORMAL METHODS, A PRAGMATIC MIDDLE GROUND

A variation to the formal methods described above could be the so called *lightweight formal methods* [JW96]. The term lightweight formal methods was introduced by Jackson and Wing in 1996 [JW96]. They advocated for the creation of a formal method approach which emphasized partiality: partiality of language, of modeling, of analysis and of composition.

This emphasis on partiality would allow to make trade-offs amongst the different partiality axes depending on the context the method is used. For instance, the partiality of language balances the trade-off between the expressiveness of the language versus the tractability of the analysis. E.g., a language which allows for the modeling of web security protocols [Kum14] is less expressive than a specification language such as Z [Spi89] but it allows for tractable, albeit partial, analysis of these protocols whereas Z is, by itself, not analyzable. Next to that, a specification language specific for web security protocols probably is much easier to pick up by security domain experts than a very generic specification language.

A lightweight formal method could thus be less expressive and maybe not as widely applicable as a traditional formal method but would be more cost effective and better suited for a certain problem domain. Although such a method would not be 'fully formal' in the sense that it would give full correctness guarantees but it might be more easy to teach and use and would give a higher level of correctness guarantee than would be achieved using traditional testing techniques. Or, to put it in the words of Jackson and Wing: "A surgical laser [...] produces less power and poorer coverage

than a light bulb but it makes more efficient use of the energy it consumes and its effects are more dramatic”.

The application of a lightweight formal method thus offers practitioners a pragmatic choice along different axes of partiality such as completeness of analysis, completeness of specification and expressiveness of the specification language. This idea of partiality offers an attractive point of departure for our research since it offers ways to influence the *depth* and *breadth* of the applied method. We are especially interested in how these different axes of partiality influence the design and verification of enterprise software systems.

The broader question that we are interested in investigating in this thesis is thus:

General Research Question (GRQ)

What is the impact of different choices along the axes of partiality on the design and verification of enterprise software systems using lightweight formal methods?

1.5 RESEARCH CONTEXT

We performed our research in collaboration with the ING bank: a large and international operating, Dutch bank. Since the early 1960's the ING bank has been automating its business processes. What started with the automation of money transferal to facilitate companies to pay their employees in an automated manner, has resulted in an enterprise ran by software. Anno 2022 almost all facets of ING's business is governed by software with thousands of unique applications running and performing all kinds of tasks: from sales to mortgages and from trading to account management.

These applications are the result of decades of software development. Since computer languages and development techniques have changed over time, so have the used languages and techniques at ING. The result is an application landscape that contains a plethora of different languages, techniques and ideas. These applications together make up the system as a whole. Keeping the software in such an enterprise consistent and evolving is challenging. In this light we explore the creation and use of lightweight formal methods that offer support to this process.

Although many of our work was inspired by challenges faced by our collaborating partner, our work is more generally applicable. The problems faced by the ING bank are not unique, they are faced by many large enterprises as has been summarized by Charette [Cha05].

There is no agreed upon definition of what makes a formal method lightweight [ZSR⁺18]. This is complicated even further by the fact that the term *lightweight* is used for two different categories: for formal methods that were specifically created to be lightweight (e.g., Alloy [Jac12]) and for ‘traditional’ formal methods that can be used in a lightweight fashion (e.g., with VDM [AL98]). The latter category is also sometimes characterized as *light touch* formal methods [Simo6]. Listing all different lightweight formal methods is thus very difficult. In the overview to follow, we chose to highlight some methods that we found are often referred to as being lightweight or used in a light touch manner.

Alloy The first method that used the term lightweight formal method was *Alloy* [Jaco2a; JSS00]. Alloy was created by Jackson in the early 2000’s. The novelty of Alloy was that its input language was based on relational logic and that it allowed for automatic bounded analysis using an intermediate representation and tooling which later became *Kodkod* [TJo7b]. The use of relational logic made it especially suitable to describe problems with structural elements. Alloy was primarily positioned as a design and verification tool for key-parts of a systems design and has since its inception been used for a broad range of designs and problems (e.g file-systems [KJo9], network protocols [Zav17] and security policies [PSK⁺11]).[‡] Alloy is still under active development with new extensions to the tool and language being added.[§]

TLA+ An example of a language and technique often referred to as lightweight is TLA+ and its model checker TLC [Lam99; Lam02]. Although not specifically created as a lightweight formal method, its use is often seen as lightweight since it can be picked up with relative ease. TLA+ originated at the same time as Alloy and has since its inception been used to model and verify many (parallel) algorithms. Like Alloy, TLA+ also emphasizes ease of writing specifications and practical verification using a bounded model checker. Since the analysis is bounded (partial), it does not formally proof algorithms correct, but in practice it does find many inconsistencies which would be hard to find using traditional testing techniques [NRZ⁺15].

VDM The Vienna Development Method (VDM) is one of the earliest formal methods originating at IBM in Vienna in 1976 [BJ]. Originally created as a rigorous method for defining the operational semantics of the languages PL/1 [BBH⁺74] it has been used since then for the formalization of many other systems. Over the years, the method

[‡]<http://alloytools.org/citations/case-studies.html> holds an collection of case studies using Alloy.

[§]The newest version of Alloy, version 6, offers the possibility to naturally encode *state transition problems* as pioneered in the earlier Alloy extension Electrum [BCC⁺18]

has also been used in a more lightweight manner. For instance, Agerholm et al. report on a number of industry cases where a lightweight version of VDM was amongst other things used to specify the doors of a metro [AL98]. Droschl et al. report on the specification of a security system module using lightweight VDM [DKS⁺00].

1.6.1 *Examples of Lightweight Formal Methods in Enterprise Software*

Using lightweight formal methods for designing enterprise software is not new either. There are some reports on the use of lightweight formal methods during the design and development of enterprise software. We will highlight some of them.

Lightweight formal methods at Amazon Newcombe et al. report on their experience of using TLA+ (and its accompanying model checker TLC) at Amazon [NRZ⁺15]. They describe how TLA+ is used at Amazon to specify and verify complex algorithms used in large distributed database, storage solutions and concurrency algorithms. According to the authors they had little trouble learning TLA+ without additional training. They report that making use of TLA+ uncovered numerous subtle bugs in software designs that would not have been discovered using traditional reviewing and testing practises. Next to that, they note that having a formal specification of a design helps during the evolution of the software itself to both try-out refactorings and optimizations and to help new engineers to get a better understanding of the software.

Another example of lightweight formal methods at Amazon was described by Bornholt et al. They describe the use of lightweight formal methods in the design and implementation of ShardStore, a highly distributed key-value store [BJA⁺21]. For this they used a combination of existing techniques that in their view can be seen as a lightweight formal method. First, they implemented reference models of ShardStore containing the expected semantics. This reference model was created in the same language that was used for the implementation making it easier for engineers to pick up. Second, the reference models are used in checking whether the implementation is conform. These checks are performed using a combination of property based testing [CH00] and stateless model checking [MQ08]. Using this methodology they prevented 16 potential bugs, including subtle concurrency and crash consistency problems, from reaching production

The Merode method Another notable method that can be seen as a lightweight formal method in enterprise modeling is *Merode* [Sno14; SMD03]. Merode was developed in the early nineties as a object oriented system development methodology [DSD92] The method was specifically created for the modeling and construction of enterprise software systems. Its unique selling point is that it exploits the so called *existence dependency* relation between different objects. This existence dependence relation is a

formal contract which enforces the life-cycle (creation, modification and deletion) of an object. A case in point is for instance that in a library information system the 'Loan' object is existence dependent on a 'Book' object: you can not borrow a book that does not exist. These domain models can be composed in Information System Services which in turn be composed in so called Business Process Services. In the end, Merode allows for the automatic generation of software (using the correct-by-construction principle) based on the defined models.

1.7 QUESTIONS ALONG THE DIFFERENT AXES OF PARTIALITY

As stated in our general research question we are interested in the trade-offs the different axes of partiality have on specification and verification methods for enterprise software. In the coming sections we define our research questions in the context of these different axes.

1.7.1 *Partiality in Language*

According to Jackson et al. specification languages need to be designed with analysability in mind. They state that: “[Analysis] tools designed as an afterthought can provide only weak analysis” [JW96]. The more general a specification language is, the harder it will be to perform deep, automatic analysis. This means that creators of a specification language must balance the expressiveness of the language with the ability to perform automatic analysis. The question is how this relates to the domain of enterprise software. We thus formulate the question:

Research Question 1: Partiality in Language (RQ1)

How can we design specification languages such that they are expressive enough to specify problems in the enterprise domain while still able to perform automatic analysis?

1.7.2 *Partiality in Modeling*

The investment needed (both in time and expertise) to fully formalize a system is in most cases not in balance with its gains. Jackson et al. argue that one should only model those parts of which the merit of formalization outweigh the cost [JW96]. An important question to ask is which risks are mitigated by the formalization.

On the other hand we know there is value in writing specifications of systems, even if no verification is performed (see e.g. [GRSo4; HA00]). Writing down system specifications forces specifiers to think about the details which are required for the needed system. Following this process leaves little room for hand waving. Even by

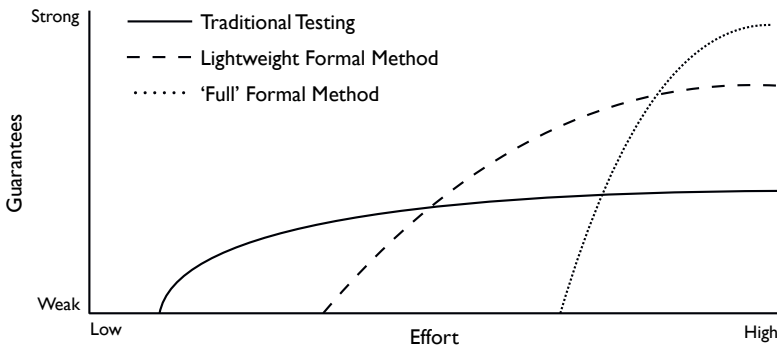


Figure 1.1: Assessment of Effort versus Guarantees of different methods.

merely writing down the specification is known to already discover many design flaws and other problems that would otherwise arise later on in the process (e.g. [DKS⁺00]). This gives rise to the following question:

Research Question 2: Partiality in Modeling (RQ2)

How can we manipulate the cost of modeling for different parts of an enterprise system while preserving the positive impact of specifying?

The reasoning here is that by allowing different levels of formality within the same specification a specifier could decide to apply certain verification techniques for parts of the specification while other parts are merely specified (without any automatic verification performed). This would give a specifier the advantages that come from “merely specifying” while still allowing to perform automatic reasoning and verification techniques on parts of the specification.

1.7.3 *Partiality in Analysis*

An important part of the rationale of lightweight formal methods is that it should allow for scalable, automatic analysis [JW96]. Having a complete and decidable reasoning technique which still is tractable is nearly impossible for large, enterprise scale systems. For this reason lightweight formal methods focus on analysis techniques which can be applied in a partial manner. For instance by bounding the scope of the analysis to a subset of the problem domain. Even though these techniques might not prove the absence of bugs, they do offer a thorough way to show their presence given a bounded scope and thus offer benefit over more traditional testing techniques (see Figure 1.1 for our assessment on the different guarantees given by the different techniques).

Although there are many reasoning techniques that can be used for partial analysis, we focus on the technique used by Alloy since this was the original method specifically

designed to be lightweight. Alloy utilizes the bounded relational model finder Kodkod for automatic reasoning [TJ07b]. Kodkod takes a relation specification as input and translates this to a boolean satisfiability problem (SAT) which in turn is solved by an off-the-shelf SAT-solver (e.g., SAT4J [LP10]). Although Kodkod is able to reason on non-relational problems such as integer constraints, the encoding of these types of constraints into a boolean satisfiability problem is not the most efficient. These type of constraints do however occur often in our problem domain of financial systems. The questions that thus arises is:

Research Question 3: Partiality in Analysis (RQ3)

How can we extend the current state of the art in relational model finding in such a way that it is possible to efficiently reason about other theories such as integer arithmetic?

The rationale would be that by extending the current state-of-the art, analyzing mixed theory problems would become more efficient and thus allowing to analyze a larger part of the problem domain in the same amount of time. This would further increase our confidence of the correctness of the specification.

1.7.4 *Partiality in Composition*

According to the definition of Jackson et al. modeling a large system will most probably result in many partial specifications. Each written for its own purpose or specific aspect of interest. In such a web of separate specifications it is hard to reason about cross-cutting properties such as consistency. This gives rise to the question how to combine these specifications in such a way that some automatic reasoning can be performed. A way forward would be to allow users to specify these compositions in an ad hoc manner allowing for different compositions to be made during the specification process. Such a solution would require a specification language which would allow for these bespoke compositions. We formulate the following question:

Research Question 4: Partiality in Composition (RQ4)

How can we lift the problem of composition to the language level such that the user is able to specify different compositions during specification and analysis?

1.8 ORIGIN OF THE CHAPTERS

In this section we will list the origin and contributions of the coming chapters in this thesis. Chapters 2–5 are all published papers at different conferences and workshops.

We will highlight which of our defined research questions are addressed in the different chapters.

Chapter 2. On the Design of the Rebel Specification Language and its Application inside a Bank

J. Stoel, T. v. d. Storm, J. Vinju, and J. Bosman. “Solving the bank with Rebel: on the design of the Rebel specification language and its application inside a bank”. In: *Proceedings of the 1st Industry Track on Software Language Engineering*. 2016, pp. 13–20

In this chapter we introduce the Rebel specification language and Integrated Specification Environment (ISE) within the context of the ING bank. We describe the requirements that underlie the Rebel specification language and how it can be translated and checked by an SMT solver. Lastly it contains an initial evaluation on the use of this ISE within the bank. In this chapter we explore Research Question 1, partiality in language, as we experiment with a specific specification language design for our collaborating partner. The thesis author was the main author of this chapter.

Chapter 3. AlleAlle: Bounded Relational Model Finding with Unbounded Data

J. Stoel, T. van der Storm, and J. J. Vinju. “AlleAlle: bounded relational model finding with unbounded data”. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 2019, pp. 46–61

This chapter contains the description of AlleAlle, a relational bounded model finder.

We introduce the language, its semantics and the translation of the language to SMT. AlleAlle is inspired by other relational modeling language like Alloy and Kodkod but it targets a different solving engine. It translates to SMT instead of to SAT. Because of this the AlleAlle model finder is able to directly reason about non-relational theories like integer arithmetic. We evaluate both the performance as well as the expressiveness of AlleAlle.

In this chapter we explore Research Question 3, partiality in analysis, as we extend on the current state-of-the-art in relational model finding. The thesis author was the main author of this chapter.

Chapter 4. Constraint-based Run-time State Migration for Live Modeling

U. Tikhonova, J. Stoel, T. Van Der Storm, and T. Degueule. “Constraint-based run-time state migration for live modeling”. In: *Proceedings of the 11th ACM*

SIGPLAN International Conference on Software Language Engineering. 2018, pp. 108–120

This paper won the *Best Software Engineering Technology 2018* award appointed by VERSEN (the Dutch National Association for Software Engineering).

In this chapter we introduce a language, Nextep, and technique that lets users describe runtime state meta-model, a technique that can be used in the context of Live Modeling. The technique automatically updates the runtime state whenever the underlying runtime meta-model is changed. This is realized by encoding the described meta-model and runtime state as AlleAlle problems. The AlleAlle model finder is then run to find a fix to the runtime state that is minimal to a chosen heuristic. To find this minimal model the AlleAlle model finder is instrumented with a *minimization criteria* that forces the model finder to look for a minimal model.

This chapter contains another use-case for the model finder AlleAlle. It also explores Research Question 1, partiality in language, as we define a language which balances live modeling requirements with the possibility for automatic analysis and repair. This chapter was done in collaboration with co-authors. The thesis author contributed to the idea, design, implementation and benchmarking of the solution.

Chapter 5. Modeling with Mocking

J. Stoel, T. van der Storm, and J.J. Vinju. “Modeling with Mocking”. In: *2021 IEEE 14th International Conference on Software Testing, Validation and Verification (ICST)*. 2021

In this chapter we introduce the notion of *Mocking to Model Checking*, allowing the user to write specifications for the complete system while still being able to check parts of the system in isolation. Mocking originated as a testing technique of object oriented systems and allows the user to test objects in isolation to other objects. We describe this mocking technique in the context of the Rebel2 specification language, a language inspired by the specification language introduced in chapter 2. The chapter contains the formalization of two introduced language constructs for mocking, namely: *mock* and *forget*. Next to that it contains an evaluation of this mocking technique on two case studies, one from the automotive industry and one from the banking domain. We find that the language is expressive enough to specify both case studies and that mocking enables the (partial) checking of these specifications where checking without mocking quickly results in long checking times.

In this chapter we explore research questions 2, partiality in modeling and 4, partiality in composition. The thesis author was the main author of this chapter.

1.9 SOFTWARE ARTIFACTS

During our research we have created a number of software artifacts. All these artifacts are available as open source and created using Rascal and the Rascal Language Workbench [KvdSV09]. The list of artifacts is as follows:

- Rebel (v1) [Sto16]
- AlleAlle [Sto19a; Sto19b]
- Nextep [Sto19c; STvdS⁺19]
- Rebel (v2) [Sto21]

The design and implementation of the different artifacts is described in more detail in Chapter 6.

Abstract

Large organizations like banks suffer from the ever growing complexity of their systems. Evolving the software becomes harder and harder since a single change can affect a much larger part of the system than predicted upfront. A large contributing factor to this problem is that the actual domain knowledge is often implicit, incomplete, or out of date, making it difficult to reason about the correct behavior of the system as a whole. With Rebel we aim to capture and centralize the domain knowledge and relate it to the running systems.

Rebel is a formal specification language for controlling the intrinsic complexity of software for financial enterprise systems. In collaboration with ING, a large Dutch bank, we developed the Rebel specification language and an Integrated Specification Environment (ISE), currently offering automated simulation and checking of Rebel specifications using a Satisfiability Modulo Theories (SMT) solver.

In this paper we report on our design choices for Rebel, the implementation and features of the ISE, and our initial observations on the application of Rebel inside the bank.

2.1 INTRODUCTION

The ING bank is an organization with a long history and was among the first Dutch companies that started automating their processes. In the past decades many different systems on different technologies were created to support the ever growing need for process automation. With every new automated service and with the growing use of these services the demand on these systems also grew quickly. During these decades the underlying technologies in which new systems were implemented changed while often the underlying problem domain did not. This resulted in an application landscape with numerous applications implemented in different technologies running on different platforms.

Reasoning about the impact of change or the introduction of new features in such a large and technologically scattered application landscape is hard. Especially since the description of the domain knowledge which is captured by these applications is often missing, out of date, or incomplete. When the domain knowledge is captured it is written down in informal documents like Word files or Excel sheets without connection to the implemented application. Changing the software becomes a labour intensive task relying on the tacit knowledge of the people in the organization. As a

result, the ability to predict and control the correctness [Mey85], cost-of-ownership, performance and reliability is compromised.

In a public/private partnership between our research institute and the ING bank we set out to improve the quality of communication between stakeholders, to simplify the design and implementation of products and services using lightweight formal methods [Jac01]. The initial result of this collaboration is the Rebel specification language and its ISE. Rebel is built using of the language workbench Rascal [KvdSV09], and employs the state-of-the-art SMT solver Z3 [DB08] for simulation and checking. In this paper we describe the current design and initial observations of Rebel within the bank.

The contributions of the current report can be summarized as follows:

- We provide a description of the requirements and design of Rebel (Section 2.3);
- We sketch how Rebel specifications are translated to SMT formulas for formal analysis and simulation (Section 2.5).
- We show how the simulation of specifications can be used to test existing applications (Section 2.6).
- We report on initial observations in applying Rebel and its ISE inside the bank (Section 2.7).

Ultimately Rebel specification could serve as the base for new applications.

2.2 BACKGROUND

Based on discussions we had with various stakeholders within the bank we identified four challenges faced by the bank.

Dispersed Functionality. The current application landscape of the bank contains approximately 1400 applications with many interactions between them of unknown scale. This landscape is the result of nearly 50 years of software evolution within the bank incorporating many different technologies, frameworks and design styles. Some of the applications have overlapping functionality, but it is often unclear whether they behave similarly under equal conditions. As a result, changes to the software require extensive, labour intensive testing.

Scattered and Implicit Domain Knowledge. Which applications should encode which part of the domain is informally documented (if at all) and scattered across the landscape. There are many partial requirements documents, ranging from written documentation and presentation slides to Excel sheets and sometimes UML diagrams like Statecharts and sequence diagrams. These partial descriptions, however, are often ambiguous, incomplete and under-specified. It is eventually up to the developers to implement the requirements to the best of their knowledge. Ultimately, the only

source of domain knowledge is the software itself, but eliciting this knowledge is a hard problem, well-known from research in reverse engineering [TP04]. As a result, questions like “What is a savings account?” or “Which operations are allowed on a savings account?” are very hard to accurately answer.

Stricter Regulations. Regulation of banking is becoming more strict. This has become even more urgent since the start of the economic crisis of 2008. As a result the accountability of the system as a whole must increase. Currently, questions like “Why did this transaction fail?” are hard to answer and require intensive manual labour like mining log files to trace calls through large parts of the application landscape.

Implicit Quality Assurance. There exists a conceptual gap between product owners—domain experts that are responsible for a specific product—and development teams. Currently the way to check whether development teams have implemented requested features correctly is by demonstrating the actual software itself or a prototype of it. It is then up to a product owner to decide whether the implemented feature meets its specifications. Since the specification are often informally documented there is currently no automated way to check whether the software meets this specification. Next to this it can be hard for product owners to find deficiencies in the product that are caused by under-specification. In other words, a product owner currently does not have many tools that help making this decision.

2.3 DESIGN OF THE REBEL LANGUAGE AND ISE

The design of the Rebel language and ISE is guided by the challenges that are described in the previous section. In the coming section we will elaborate on the design choices. How these choices are reflected in the language and ISE is delayed to later sections.

Centralized and Unambiguous Specifications. To tackle the challenge of scattered and implicit domain knowledge we designed Rebel as a formal specification language in which it is possible to represent the essential characteristics of financial products at a very high level of abstraction. The language focusses on capturing domain knowledge and omits any details of technical implementation. Rebel is designed as a domain specific language (DSL) for the banking enterprise domain. This means that it should be possible to capture all parts of banking. For example, banking is both about financial transactions and customer relations. Both domains can be described with Rebel.

Increase Collaboration between Product Owner and Development Team. It is important that both product owner and development team have a thorough under-

standing of the product that was specified. For instance, a product owner must decide whether or not a specification is complete regarding the desired functionality. To check this so called external consistency of the product specification [SMD03] the ISE offers a simulation environment as a means for rapid prototyping. Next to this it should be possible to visualize Rebel specifications in other formalisms like UML Statecharts, a formalism which is well known to the product owners of the bank. In our vision both product owner and development team take part in the specification process. By making use of techniques like simulation and other visualization methods we aim to minimize the possibility for miscommunication and thus ultimately, the inception of the wrong software.

Practical Automatic Reasoning. In light of stricter regulations reasoning on the level of specification is required. For instance in the Netherlands the legislator prohibits a person under the age of eighteen from buying certain financial products. Being able to check whether the specification would allow or disallow this behavior is of great value to the bank. This kind of automatic reasoning should still be practical in its use, meaning that a user of Rebel should be able to check this kind of properties by the push of a button in a reasonable amount of time. In the Rebel ISE we try to balance completeness of reasoning with the response time of the reasoning process.

2.4 REBEL SPECIFICATIONS EXPLAINED

We introduce Rebel using an example specification. For explanatory reasons we chose a simplified version of the real ING savings account.*

2.4.1 *The SimpleSavings Account*

The SimpleSavings account can be opened by a customer provided that the customer deposits more than or equal to 50 euro into the account on opening. After the account is opened the customer can deposit and withdraw money. Next to depositing money, money can also flow into the account when interest is received. The amount of interest that a customer gets is variable but it never exceeds a fixed percentage. This percentage differs over different saving accounts but for this SimpleSaving account it is fixed at 5%. In extreme cases, for instance when the customer is under suspicion of criminal behavior, the account can be blocked. This means that no money can be withdrawn from, or deposited to the account. In the end, the account can be closed provided that the remainder of money has been taken out of the account. When it is closed the account finally ends up in a state without any possible further interactions. It is invariant for every type of savings account that the balance is always positive. In other words, it is not allowed to overdraw a savings account.

*See Appendix B.1 for the complete syntax definition of REBEL.

```

1  specification SimpleSavings {
2      fields {
3          accountNumber: IBAN
4          balance: Money
5      }
6
7      events {
8          openAccount[minimalDeposit = EUR 50.00]
9          withdraw[]
10         deposit[]
11         block[]
12         unblock[]
13         interest[maxInterest = 5%]
14         close[]
15     }
16
17     invariants { mustBePositive }
18
19     lifeCycle {
20         initial init -> opened: openAccount
21         opened      -> opened: withdraw, deposit, interest
22                   -> blocked: block
23                   -> closed: close
24
25         blocked    -> opened: unblock
26         final closed
27     }
28 }

```

Figure 2.1: Rebel specification of a SimpleSavings account. The events and the mustBePositive invariant are explained in more detail in Section 2.4.3.

2.4.2 Business Entities as State Machines

The Rebel implementation of the SimpleSaving account is shown in Fig. 2.1.

Each specification consists of four (optional) parts: state variables, event declarations, invariants and life cycles. The `fields` section describes state variables of this entity and their types (e.g., `balance`, `accountNumber`). Since Rebel has been designed specifically for the banking domain, some of the types are specific for this domain. For example, Rebel has built-in types for `Money`, `Currency` and `IBAN` (unique European bank account number), next to the standard types for booleans, numbers and strings. With this design we aim to maximize the range of banking domain problems which can be expressed, while using values which are natural to domain experts, and without forcing them to formalize “trivial” details (e.g., uniqueness of `IBAN`).

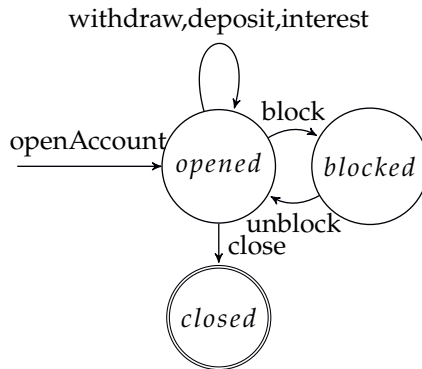


Figure 2.2: Graphical representation of the life cycle of the SimpleSavings example shown in Fig. 2.1

The `events` section contains all possible transition triggers (e.g., `openAccount`, `withdraw`, `deposit`, etc.). The `lifeCycle` section defines a state machine for each instance of the specification. Fig. 2.2 depicts the state machine of a `SimpleSavings` instance. The pattern for a transition $s_1 \rightarrow s_2 : e_1, \dots, e_n$ should be read as a transition from s_1 to s_2 is possible via any event in e_1, \dots, e_n . For instance, it is possible to block an account that has been opened, which changes the state of the savings account to state `blocked`. A transition fires if and only if it is enabled by the life cycle and its conditions are satisfied (see section 2.4.3). The `initial` and `final` keywords mark the initial and final states of the machine.

Finally, the `invariants` defined for an entity capture predicates that always have to be true. For instance, it should not be possible to overdraw from the savings account, given the invariant `mustBePositive`; it is up to the conditions of the `withdraw` event to satisfy this invariant.

2.4.3 Declaring Events and Invariants

Note that Fig. 2.1 does not show the definition of the pre- and post conditions and invariants. These are specified elsewhere to promote reuse of events and invariants for specifying other, similar business entities, and for making Rebel specifications more concise. As an example, consider a possible definition of the `openAccount` event, shown in Fig. 2.3. Event definitions have two sets of parameters: configuration parameters (enclosed in `[...]`) and transition parameters (enclosed in `(...)`). Configuration parameters are bound at design time, and have default values. The default values can be overridden when an event is included in a specification. For instance, the configuration parameter `minimalDeposit` with the default value of `EUR 0.00` (Fig. 2.3, line 1) is bound in `SimpleSavings` (Fig. 2.1, line 7) to be `EUR 50.00`. With the use of

```

1  event openAccount[minimalDeposit: Money = EUR 0.00]
2    (accountNumber: IBAN, initialDeposit : Money) {
3
4    preconditions {
5      initialDeposit >= minimalDeposit;
6    }
7
8    postconditions {
9      new this.balance == initialDeposit;
10     new this.accountNumber == accountNumber;
11   }
12 }

```

Figure 2.3: Definition of the openAccount event

these configuration parameters it is possible to reuse the openAccount event in the specification of other types of ING saving accounts.

Events can refer to entity fields using the keyword `this`. This notation is borrowed from object-oriented languages and refers to the current instance of the specification. The keyword `new` refers to the value of the variable in the post-state, after the transition has fired. This distinction is necessary to express logical and arithmetic constraints between the state of an instance before and after each transition. For instance, when withdrawing money the post-state of the balance state variable would be expressed using the current value (`new this.balance == this.balance - amount`). Semantically, all the constraints written in the preconditions must hold for the instance of the specification to make the transition and after the transition it is asserted that post-condition holds.

Invariants are also specified separately and they specify predicates that must be true at all times. For instance, the `mustBePositive` invariant exists to assert that the balance of the `SimpleSavings` is greater than or equal to EUR 0.00 in all reachable states:

```

invariant mustBePositive { this.balance >= EUR 0.00 }

```

2.5 SIMULATION AND CHECKING SPECIFICATIONS

Both simulation and checking share the the same underlying encoding strategy which will be explained in the next subsection. The subsections that follow will contain more details on the individual steps.

2.5.1 Overall SMT Encoding Strategy

We define the semantics of Rebel in terms of labeled transition systems (as introduced by Keller [Kel76] and popularized by Plotkin [Plo81]). The current state of the transition

system for a given specification maps to a named state of the specification, tupled with the current field variable assignments and the event parameter assignments that led to the current state. The labeled transitions map to the events and their preconditions and postconditions. The invariants are used as external specifications of expected behavior and are simply mapped to additionally asserted formulas.

Labeled transitions systems can be checked using bounded model checking [VBG⁺09]. We use an encoding of symbolic bounded model checking (with data) as an SMT problem inspired by Milicevic [MK11] and Veanes et al. [VBG⁺09]. The goal of bounded model checking is to find a reachable state in which some property of interest does not hold (e.g., a state in which some invariant does not hold). We use the bound, encoded by k , as a parameter to balance efficiency and response time in the ISE with completeness of the check and we use it also to explore the state-space in a breadth-first manner for simulation purposes.

In bounded model checking, the *initial state* s_0 is constrained by some function: θ . For Rebel the semantics of the function θ constrains the initial state to represent an uninitialized specification.

Next, the *transition function* that constrains the valid transition from one state s_{i-1} to the next s_i is captured by $\rho(s_{i-1}, s_i)$. This means that in a valid trace—a chain of valid transitions from one state to the next—the following formula must hold: $\rho(s_0, s_1) \wedge \rho(s_1, s_2) \wedge \dots \wedge \rho(s_{k-1}, s_k)$. For Rebel the semantics of the transition function is the exclusive disjunction of all defined events. For instance, a `SimpleSavings` specification can only transition via `withdraw` \oplus `deposit` but never both. This means that a state transition in a Rebel specification is always constrained by only one of the defined events.

Like mentioned earlier, the goal of bounded model checking is to find a reachable state in which some property of interest does not hold. This property is also known as the *safety property* and captured by the function P . We are interested in a trace in which the safety property holds in all states except for the last. More formally, the following should hold: $P(s_0) \wedge P(s_1) \wedge \dots \wedge \neg P(s_k)$. In the case of Rebel we are interested in finding states where the invariants do not hold. The safety property is defined as the conjunction of all defined invariants since a specification can contain multiple invariants.

So, using the above mapping of Rebel to SMT formulas we may verify—up to transition traces of length k —that the specified invariants hold during the entire life cycle of any entity. We may use the same mapping to infer single transition steps to run a simulator. In this case the SMT solver provides us with computations which satisfy the route from pre-condition to post-condition for every transition. Effectively, the SMT solver has then become an interpreter for Rebel specifications.

In the coming sections we will give a more detailed description of the steps that are used when translating a Rebel specification to SMT constraints.

2.5.2 Normalization

Before we simulate or check a Rebel specification it is normalized. Normalization of the Rebel specification is not only done to make the mapping to SMT formulas easier; it also partially gives semantics. After normalization a specification contains all the necessary information for the aforementioned mapping to SMT. It consists of the following steps:

1. **Inlining.** Referenced events with their configurations and invariants are resolved and inlined with the specification.
2. **Desugaring the Life Cycle.** The life cycle is desugared by strengthening the pre- and postconditions of the events with the life cycle information. To achieve this two fields, `_state` and `_step`, are added to the fields of the specification. A distinct identity is assigned to each state and event. The `_state` field holds the identity of the current state and the `_step` field holds the identity of the event that led to the current state. This way the original life cycle can be expressed by adding constraints based on the two newly added fields to the pre- and postconditions of the events.
3. **Adding Frame Conditions.** To guard the fields that are not changed by the event frame conditions are added [Jac12]. These frame conditions make sure that a field has the same value after the transition as before.

2.5.3 Bounded Checking

The goal of checking Rebel specifications is to check whether a given specification is consistent. A specification is considered consistent if the invariants hold in all reachable states. A reachable state is a state which can be reached from the initial state via (a chain of) valid transitions. Since Rebel specifications have data encapsulated and life cycles may have loops model checking without reasonable bounds could quickly suffer from the state explosion problem [CES09]. Here we describe two verification techniques we implemented for Rebel (both use a similar encoding).

Step 1: Quickly Check if the Specification is (trivially) Consistent. First we use the SMT solver to try to inductively prove that the invariants hold in all possible transitions. This is expressed using these three formulas:

1. If the initial condition holds in some state then the safety property should also hold: $\theta(s_0) \Rightarrow P(s_0)$
2. If the initial condition holds in the first state and there is a transitions possible to a second state then the safety property should also hold in the second state: $\theta(s_{i-1}) \wedge \rho(s_{i-1}, s_i) \Rightarrow P(s_i)$

3. If the safety property holds in the first state and there is a transition possible to a second state then the safety property should also hold in the second state:

$$P(s_{i-1}) \wedge \rho(s_{i-1}, s_i) \Rightarrow P(s_i)$$

If these three formulas can be proven by the SMT solver it means that the specification is consistent. In this case we report back to the user that the specification is found to be consistent. If they can not be proven it means that there might be a transition possible which leads to a state in which the safety property does hold.

This strategy can lead to false positives—a specification which is wrongly labeled as inconsistent—but never to false negatives. For instance: if the third hypothesis can not be proven it means that it is possible to construct a state in which the invariants hold, make a valid transition and end up in a state in which the invariants do not hold. However, whether the first state of the counter example is reachable from the initial state is unknown making it a potentially unreachable counter example. To find out if the counter example is actually reachable we run a bounded model check on the specification.

Step 2: Run Bounded Analysis. During bounded analysis we are interested in finding the smallest possible counter example, where smallest means in the least possible steps. The formulas that we try to prove are similar to those described in the previous section but the difference is that we now use explicit step unwinding.

The process of finding a counter example is fully automatic and incremental. We start by checking if an invalid state can be reached in one step. If not then we check if it can be reached in two steps. This is continued until a counter example is found or some k is reached[†]. More formally, we try to prove that:
$$\theta(s_0) \wedge P(s_0) \wedge \rho(s_0, s_1) \wedge P(s_1) \wedge \dots \wedge \rho(s_{k-1}, s_k) \wedge \neg P(s_k).$$

If a counter example is found by the solver the found model is translated back to the Rebel simulator which can then visualize the counter example as a trace in the simulation environment. If no counter example can be found in the given k the ISE reports to the user that the specification might be consistent. The phrasing ‘might’ is used since it still could be the case that a counterexample can be found after n steps where $n > k$.

2.5.4 Simulation

The purpose of simulation differs from checking. Where checking is done to check the internal consistency, simulation is used to check the external consistency [SMD03]. Fig. 2.4 shows a screenshot of the implemented simulator in the Eclipse IDE. Using the simulator the user can quickly check whether the created specification behaves as

[†]In our implementation we have chosen to work with a configurable timeout given to the SMT solver instead of some fixed k . This choice is practical by nature, we want to control the maximum time spent waiting by the user.

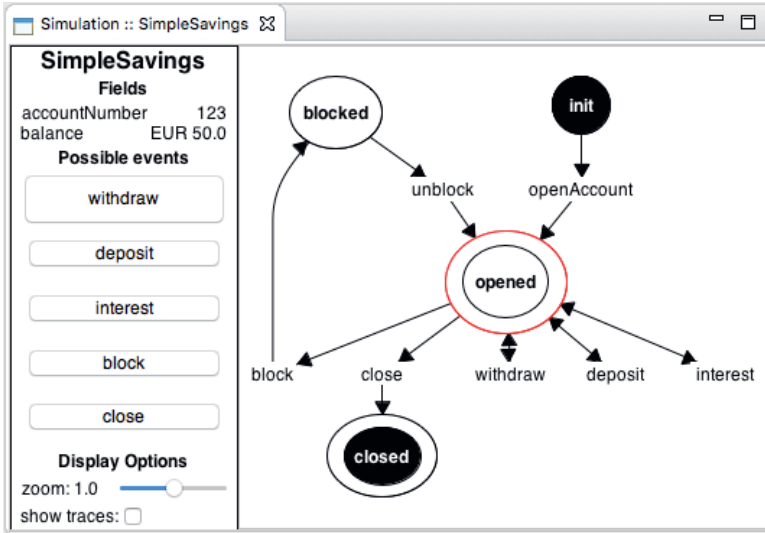


Figure 2.4: Screenshot of the Rebel simulator. The right side shows a graphical representation of the life cycle. The current state is highlighted by a red ellipse. The top left side contains the current values assigned to the fields, and the bottom part displays buttons for each possible transition that can be fired from the current state.

(informally) expected. Where checking is about reasoning about all possible traces, simulation is about reasoning about individual steps. This can be implemented using the similar strategy as described in the previous section. As mentioned earlier, when simulating we effectively use the SMT solver and our encoding as an interpreter of Rebel specifications.

Translate a Single Step to SMT. We translate the event the user wants to execute to SMT formulas. The transition function, $\rho(s_1, s_2)$, contains the pre- and postconditions of the to be executed event. The current values of the simulated specification are translated as constraints on the current state, s_1 , and the user is asked to provide the data values for the transition parameters of the chosen event transition.

Next, the solver is asked whether it can indeed ‘make the step’. This means that we check two things:

1. Whether it is possible to satisfy the constraints of the selected events given the current state and actuals of the transition parameters of the event: $\rho(s_1, s_2)$
2. Whether the invariants (the safety property) hold in the resulting state: $P(s_2)$.

If it can not make the step because the first check fails we make use of the *unsatisfiability core* functionality of the solver to find out which constraints are most likely the cause

of the failure. The unsatisfiability core functionality reports an unsatisfiable subset of clauses of the asserted formulas [CGS11]. The returned constraints are mapped back to the original Rebel expressions and presented to the user. If it can not make the step because the second check fails we report back to the user which invariant is violated as a result of the step, and we roll back to the previous state so that the user can explore other options without having to restart the simulation.

2.6 PERFORMING MODEL BASED TESTING OF EXISTING APPLICATIONS

Another important aspect of the Rebel ISE is the ability to check whether an existing application behaves according to the specification. Mismatches between specification and the system under test (SUT) can point to bugs in the existing applications or to erroneous assumptions in the specification. This is essentially a model based testing approach to test existing ING applications [DJK⁺99].

To implement this functionality we use the traces that were described in Section 2.5. By automatically checking whether the SUT accepts the trace as a valid execution trace we can check whether the SUT behaves similar as the specification. To playback the steps in a trace on the SUT every transition is split into three steps:

1. Check whether the current state of the SUT conforms to the current state in the trace (pre-transition check)
2. Ask the SUT to perform the actual operation according to the trace (transition check)
3. Check whether the new state of the SUT conforms to the new state in the trace (post-transition check)

To implement the above steps we map every event declared in the specification to an operation in the SUT (to perform the transition check). Next to that we map fields of the specification onto the state of the SUT (to perform the pre- and post-transition checks). A requirement on these mappings is that all events and fields of the specification are completely mapped onto the SUT (otherwise the trace effectively can not be played back). Our first prototype used SOAP services provided by the SUT and performed the operations by sending SOAP messages and by checking whether the received responses were conform to the trace.

Currently, it is possible to perform the described testing interactively using the simulation described in Section 2.5.4. Every step made in the simulation is automatically also performed in the configured SUT. Any difference between the simulation and the SUT is then displayed in the simulation showing which values differ between the two. Expanding this functionality to also work completely automatically using a given trace is future work but should be straightforward to implement.

2.7 APPLYING REBEL INSIDE THE BANK

We implemented a prototype of the Rebel ISE and tested its implementation inside the bank. As a first test case we specified saving accounts [Pet14]. Fourteen (out of seventeen) types of ING saving accounts were specified. These saving accounts were built up out of fifteen distinct events. With the use of the configuration parameters these events could be reused across different types of savings accounts.

The verification tool showed an unexpected counter example where the `mustBePositive` invariant would not hold, caused by the —unlikely but real— possible circumstance of a negative interest rate.

Using the prototype of the model based testing tool we found a difference between the specification and an existing system where the existing system allowed for accounts with incorrect account numbers.

During the use of Rebel inside the bank we observed that our initial assumption that Rebel syntax would be understandable for product owners was incorrect. We informally evaluated the understandability of Rebel specifications by asking a handful of product owners whether they understood the specification. Some had more trouble than others in performing this task. When faced with a manually written document containing similar specifications and visualized using UML Statechart diagrams the same product owners were able to understand the specifications. This led us to develop transformations from Rebel specification to both natural language documents and interactive UML Statechart visualizations to increase the ease of understanding amongst product owners. A more thorough evaluation on the understandability of Rebel specification is left as future work.

2.8 RELATED WORK

Formal Specification Languages. Rebel is inspired by other formal methods. We will discuss Alloy [Jaco2a] and B [Abr96] since Rebel has similarities with both.

Alloy is a specification languages based on relational logic. Alloy is positioned as a lightweight formal method [Jaco1] meaning that instead of demanding rigorous proofs of the specifications it uses the bounded model finder Kodkod [TDo6] to analyze the specification and gives counter examples if the assertions made in the specifications do not hold. Rebel is also a lightweight formal method in the sense that it uses similar bounded analysis of its specifications. Unlike Alloy, Rebel does allow other theories to be used next to relational logic. For instance, a defined Rebel parameter can be of type **Integer** allowing for all the usual arithmetic expressions to be used in the pre- and postconditions. Next to this Rebel and Alloy handle state differently. Alloy is a general purpose specification language allowing for different modeling paradigms. It is possible to model state based systems in Alloy but this is not a builtin, meaning that users should take special care when modeling state based

systems. In Rebel State is a first class concern making it hard for other modeling paradigms to be used.

B is a formal method in which abstract machines play a central role [Abr96]. It uses first order logic and set theory to define operations on abstract machines. B was built with code generation in mind. To achieve this B uses a process called specification refinement. In every new refinement step the user adds more detail to the specification. The last level of refinement is the actual code which can be executed. On every refinement level B requires users to provide proof that the refinement is correct. Most of these proofs can be obtained automatically but, if not, they must be provided by the user. This proof obligation is the biggest difference with Rebel. Rebel follows the same philosophy as Alloy. Requiring full proofs can be experienced as ‘too heavy’ by our intended users.

Enterprise Modeling. Modeling enterprise systems is a well known topic in both research and industry (i.e. [DGW⁺14]). There are many approaches, but here we highlight only one, MERODE, since it has a unique formal analysis component. MERODE is a domain modeling approach for enterprise systems [SMD03]. MERODE defines static entity-relationship (ER) models and a dynamic model based on a process algebra. By combining the notion of existence dependency [SD98] in the ER model, with the process algebra of the dynamic model, an enterprise model can be checked for deadlocks [DS95]. Although MERODE offers some formal analysis techniques it was not build with verification in mind. For instance, MERODE allows for the definition of (class, attribute and method) constraints (in an OCL like syntax) but there is no method to check whether certain assertions will hold. It is left to the user to transform these constraints to meaningful implementations using model-to-source transformations.

DSLs and Finance. Domain-specific languages (DSLs) have a long history in the domain of finance[‡]. One of the earliest financial DSLs is RISLA [AvDR95]. The language was designed to capture the nature of interest products offered by banks. One of the findings of the authors was that financial engineering was extremely suitable as an area to apply formal methods, because financial damage inflicted by incorrect system behavior can be very severe.

The difference with Rebel and other financial DSLs is the scope of the problem domain that is covered by the DSL. While Rebel focusses on the whole of the banking enterprise, other financial DSLs, like RISLA, are specifically created to work on one specific financial problem domain.

[‡]See [CFS] for an overview

Formal Methods and Finance. Gimblett, Roggenback and Schlingloff present a case-study on how to formally specify the international Electronic Payment System (ep2) standard in CSP-CASL [GRSo4]. With the formal specification they were able to identify a number of deficiencies in the standard. As a source for the formalization they used the informal documentation of the ep2 standard which was mostly text-based, augmented with UML-like diagrams. They found that it was easy to formalize high level descriptions but when it came to the details the standard was often found lacking.

Gimblett, Roggenback and Schlingloff show that there is value in formally specifying financial standards. It is not their aim to provide a method for specifying these types of standards. With Rebel we aim to provide such a method.

2.9 CONCLUSION

In this paper we presented the Rebel language and its ISE, an integrated specification environment for defining financial enterprise systems. With the use of the Rebel ISE we were able to formally specify banking products like saving accounts. By using a mapping of the Rebel language to SMT formulas it is possible to simulate and check Rebel specifications. Simulation is useful for checking the external correctness of the specification ('does the product behave as I expected') and checking is useful to check the internal correctness of the specification ('do the specified invariants hold'). The mapping to SMT uses the same strategy for both simulation and checking using a bounded model checking encoding of the Rebel specifications.

Our first impression of the use of Rebel inside the bank is that formal specifications help in translating vague and ambiguous product description in precise product specifications. Simulation helps as an early prototyping mechanism with which users can verify whether the specified product is complete regarding its functionality. Checking helps users verifying internal consistency. It does not allow for traces where the system ends up in a state in which the defined invariants do not hold.

Next to this we observed that transforming the specifications into documentation that closely resembles the current documents that are written by hand seems to help understanding. As future work we would like to more thoroughly evaluate the understandability of Rebel specifications so we can further improve the communication between stakeholders using these specifications.

Based on these initial results we are now further investigating the optimization of the model checking and simulation processes, adding features of (parallel) composition of entities and communication between entities from Rebel specifications. Meanwhile, the bank has invested to produce more Rebel specifications of their products and services as they already see the benefit of having an unambiguous product specification as a method of communication.

ALLEALLE: BOUNDED RELATIONAL MODEL FINDING WITH UNBOUNDED DATA

Abstract

Relational model finding is a successful technique which has been used in a wide range of problems during the last decade. This success is partly due to the fact that many problems contain relational structures which can be explored using relational model finders. Although these model finders allow for the exploration of such structures they often struggle with incorporating the non-relational elements.

In this paper we introduce ALLEALLE, a method and language that integrates reasoning on both relational structure and non-relational elements—the data—of a problem. By combining first order logic with Codd’s relational algebra, transitive closure, and optimization criteria, we obtain a rich input language for expressing constraints on both relational and scalar values.

We present the semantics of ALLEALLE and the translation of ALLEALLE specifications to SMT constraints, and use the off-the-shelf SMT solver Z₃ to find solutions. We evaluate ALLEALLE by comparing its performance with KODKOD, a state-of-the-art relational model finder, and by encoding a solution to the optimal package resolution problem. Initial benchmarking show that although the translation times of ALLEALLE can be improved, the resulting SMT constraints can efficiently be solved by the underlying solver.

3.1 INTRODUCTION

In the last decades relational modeling and model finding has been used to solve problems in a wide range of domains, from security [BCR⁺14], program verification and testing [GGL⁺14; KYZ⁺11], to enterprise modeling [BPH⁺13].* Since many computational problems have relational structures relational model finding has shown to be a powerful and useful method. But there is also a large class of problems that is not purely relational and requires reasoning over other attributes as well.

Consider for instance, a simple file system. This structure can be naturally expressed as a relational problem. However, adding constraints on properties like the *depth* or the *size* of file system nodes is not straightforward, or cannot be solved efficiently. In this paper we propose ALLEALLE, a language that allows users to model both the relational and the non-relational elements—the data—of their problem.

ALLEALLE combines first order logic, Codd’s relational algebra (projection, restriction, renaming and natural join) [Cod70], and (reflexive) transitive closure in

*For an overview of the different areas where relational model finding has been applied visit <http://alloytools.org/citations/case-studies.html>.

a single formalism. ALLEALLE specifications can be translated to SMT formulas which in turn can be solved by an off-the-shelf SMT solver, such as Z3 [DBo8]. We implemented these ideas in a prototype tool.[†]

Next to solving Constraint Satisfaction Problems (CSP), ALLEALLE can be used to solve Constraint Optimization Problems (COP) (cf. traveling salesman). This is achieved by extending the syntax of ALLEALLE with the ability to express *optimization objectives* on relations. These optimization criteria are added to the translated SMT formulas and can be solved using Z3's built-in optimization solver νZ [BPF15].

We perform an initial performance benchmark and evaluate ALLEALLE's expressiveness on a well-known problem in software engineering: optimal package resolution [ATC⁺11]. This problem, faced by software package managers, can be compactly expressed as a relational problem in ALLEALLE and we show that the resulting SMT formula can be efficiently solved by the underlying SMT solver.

The contributions of this paper can be summarized as follows:

- ALLEALLE, a language combining Codd's relational algebra with first order logic, transitive closure, and optimization objectives (Section 3.3).
- A translation semantics expressed by compiling ALLEALLE specifications to SMT constraints (Section 3.4).
- Initial performance benchmarking of ALLEALLE, including a realistic benchmark based on the optimal dependency resolution problem (Section 3.5).

We conclude the paper with a discussion of related work (Section 3.6), and an outlook towards future work (Section 3.7).

[†]<https://github.com/cwi-swat/allealle>


```

1 File (oid:id, depth:int, size:int) >= {<f0,2,100>} <= {<f0,2,100>,<f1,?,?>,<f2,?,?>}
2 Dir (oid:id, depth:int, size:int) <= {<d0,?,?>,<d1,?,?>,<d2,?,?>}
3 Root (oid:id) = {<d0>}
4 contents (from:id, to:id) >= {<d0,d1>}
5 <= {<d0,d0>,<d0,d1>..<d2,d2>,<d0,f0>..<d2,f2>}
6
7 // Contents is a relation that goes from Dir -> (Dir+File)
8 contents in (Dir[oid as from][from] x (Dir + File)[oid as to][to])
9 // A dir cannot contain itself
10 forall d : Dir[oid] | no d[oid as to] & (d[oid as from] |x| ^contents)[to]
11 // Root is a Dir
12 Root in Dir[oid]
13 // All files and dirs are (reflexive-transitive) 'content' of the Root dir
14 (File[oid] + Dir[oid])[oid as to] in (Root[oid as from] |x| *contents)[to]
15 // All files and dirs can only be contained by one dir
16 forall f : (File + Dir)[oid] | lone contents |x| f[oid as to]
17
18 // All files have a positive size
19 forall f : File | some f where size > 0
20
21 // The size of a dir is the sum of all files that are transitively part of this directory
22 forall d : Dir |
23   let containedFiles = (d[oid][oid as from] |x| ^contents)[to][to as oid] |x| File |
24     some (d x containedFiles[sum(size) as totalSize]) where size = totalSize
25
26 // The depth of a file or directory is equal to the depth of its parent + 1
27 forall d : Dir[oid,depth], o : (Dir + File)[oid,depth] |
28   o[oid][oid as to] in (d[oid][oid as from] |x| contents)[to] =>
29     some (o[oid as to] x d[depth as parentDepth]) where (depth = parentDepth + 1)
30
31 // The depth of Root is 0
32 some (Root |x| Dir) where depth = 0
33
34 // Get a solution with the least number of files and directories
35 objectives: minimize (File + Dir)[count()]

```

Listing 3.1: ALLEALLE specification of a small file system, original example comes from [Tor09]. [...] is projection, [...] as ...] is renaming, x is cartesian product, & is intersection, + is union, * and ^ are (reflexive) transitive closure, |x| is natural join.

3.2 ALLEALLE

ALLEALLE is an intermediate language similar to KODKOD's [TJ07a] internal model. As such it is aimed at being a target language for high-level relational modeling languages such as ALLOY [Jac12; TD06]. Instead of using SAT solvers to solve relational constraints, however, ALLEALLE leverages native data theories built into SMT solvers, such as Z3. Because of this, ALLEALLE can support constraints over unbounded data such as integers, reals, and strings, without having to encode such

data values into boolean propositions. As a result, relational specifications employing constraints over data do not suffer from exponential blow-up problems that may occur, for instance, when using fixed bit-width integers in ALLOY or KODKOD. In other words, the solving power of ALLEALLE is a super-set of that of KODKOD.

ALLEALLE is designed to be extensible. Our current implementation supports native integer constraints, but the design of the language and the translation to SMT constraints allows support for other theories (e.g., reals, strings, etc.) in the same way as the current prototype supports integer constraints. Below we illustrate how ALLEALLE combines Codd's relational algebra and unbounded data constraints using the example of a file system specification.

3.2.1 Modeling a File System in ALLEALLE

Imagine that we would like to model a new kind of file system and we want to test our design before building the new system. Our new simple file system would have the following structural constraints: it may contain both directories and files, it only has one root, there can be no cyclic dependencies and everything must be reachable from the root. The file system does not allow symbolic links (preventing cyclic references).

Next to these structural constraints we also have some non structural constraints namely, every file must have a positive size; the size of a directory derives from the size of its contents. Finally, every file and directory has a depth which encodes the distance from the root in the hierarchy.

To check these constraints we create an ALLEALLE specification that encodes the above constraints, as shown in Listing 3.1. In the next paragraphs we will explain the different parts of this specification.

Declaring relations The first part, lines 1 to 5, contains the declarations of the relations. Every relation declaration has three parts: the name of the relation, its header, and its tuple bounds. In ALLEALLE all relations are bounded meaning that all the tuples that are potentially part of the relation are defined in its upper bound.

For instance, the `File` relation on line 1 has three attributes which are defined in its header: `oid`, `depth` and `size`. The attribute `oid` is of the `id` domain while `depth` and `size` are of the `int` domain. The `id` domain is a bounded domain of arbitrary chosen labels, or *atoms*. The domain contains exactly those values as specified in the relation declarations of the specification. For instance, for this specification the `id` domain consists of `f0`, `f1`, `f2`, `d0`, `d1` and `d2`.

The right hand side of the relation declaration lists the tuple bounds. These encode the tuples that can be part of a relation. The `File` relation contains both a lower bound (the tuple set after the `>=` sign), and an upper bound (the tuple set after the `<=` sign). Every relation must have an upper bound. Lower bounds are optional.

Lower bounds can be used to encode partial solutions [TJ07b].[‡] They encode the tuples that *must* be part of every satisfying instance. In our example we see that the lower bound of the `File` relation has one tuple, $\langle \text{oid: } f0, \text{depth: } 2, \text{size: } 100 \rangle$. This means that in every satisfying instance found by the solver the relation `File` must at least contain this tuple. In other words, we specify that our file system always must have the file (identified by) `f0` with a size of 100 and two steps removed from the root (`depth: 2`).

The upper bound, on the other hand, contains the tuples that *may* be part of a satisfying instance. For the `File` relation this means that two more tuples may be part of any satisfying instance. Both of these tuples contain question marks for the `depth` and `size` attributes. These question marks—or *holes*—in the tuple definition indicate that the value can be freely assigned by the solver as long as the values satisfy the specification. Holes can only be introduced for non-`id` attributes. Attributes of the `id` domain always need a value assigned.

The lower and upper bounds of the `Root` relation (line 3) are equal to each other and contain a tuple set with only one tuple, $\{ \langle d0 \rangle \}$. When the lower and upper bounds of a relation are equal, the `=` sign is used to define the exact bound. As a result, in every possible satisfying instance this relation must contain exactly these tuples and not more. The `Dir` relation only has an upper bound (line 2). This indicates that, according to the relation definition, the empty relation is an accepted instance.

The `contents` relation (line 4–5) is a binary relation encoding which directories and files are contained by some directory. The `..` notation is a short hand notation to define a range of tuples.[§]

Declaring constraints The next part of the specification, lines 7 to 32, describes constraints on the relations. Specifications have no directionality but for clarity of the example we artificially split the constraints in two parts. The first part, lines 7 to 16, defines constraints on the relational shape of the solution. The second part, lines 18 to 32, defines constraints on the data.

Line 8 constrains the `contents` relation to be a subset of the `Dir × (Dir + File)` relation. This enforces the `contents` relation to only contain tuples of which the `id` of the `from` attribute exists in the `Dir` relation and the `id` of the `to` attribute exists in either the `Dir` or `File` relation. Without this constraint the content relation might contain junk. In other words, it might contain tuples tying non-existing directories to other non-existing directories or files. Since the relation definition does not state anything on how different relations relate to each other these associations must be supplied as extra constraints.

[‡]Lower and upper bounds in relational model finding were first introduced in Kodkod [TJ07b].

[§]The range $\langle d0 \rangle .. \langle d2 \rangle$ denotes the tuples $\langle d0 \rangle, \langle d1 \rangle, \langle d2 \rangle$. Likewise, the range $\langle d0, f0 \rangle .. \langle d1, f1 \rangle$ denotes the tuples $\langle d0, f0 \rangle, \langle d0, f1 \rangle, \langle d1, f0 \rangle, \langle d1, f1 \rangle$.

Union compatibility In Codd’s relational algebra the union (+), intersection (&), difference (-), subset (**in**), and equality (=) operators require relations to be *union compatible* with each other. This means that both relations must have the exact same header (both attribute names and associated domains). For instance, on line 12 the constraint that enforces that `Root` ‘is a’ `Dir` is expressed using the subset (**in**) operator. The header of the `root` operator on the left hand side only contains the single attribute `oid` of the **id** domain. The header of the `Dir` relation on the other hand has three attributes (`oid`, `depth` and `size`). Since the subset operator needs the relations to be union compatible we use the projection (`[]`) operator on the `Dir` relation to project the `oid` attribute out of the `Dir` relation resulting in a new relation with only one attribute of the same domain.

Transitive closure Line 10 states that no directory can contain itself expressed with the use of the transitive closure operator. Both the transitive closure (\wedge) and reflexive transitive closure ($*$) are special operators in `ALLEALLE`. They are not part of the traditional relational algebra since it is not possible to calculate such a transitive closure on relations in general [AU79].

Since `ALLEALLE` relations are bounded it is possible to implement both operators, albeit with restriction: both operators only operate on binary relations with two attributes of the **id** domain. Line 10 applies the transitive closure over the `contents` relation.

The other constraints in the first part of the specification ensure that all directories and files are reachable from the `Root` directory (line 14), and that files and directories can only be contained by one directory or none, as is the case for the `Root` directory (line 16).

The used multiplicity constraints **lone** and **some** have their standard semantics: **lone** means zero or one tuple in the relation is required, **some** means at least one tuple is required.[¶]

Restriction Lines 18 to 32 define data constraints. Line 19 states that all files must have a positive size. To express this constraint the restrict operator (**where**) is used. Using the restrict operator we can formulate constraints on the attributes of a relation. Applying the restrict operator on a relation results in another relation. To enforce that this restriction holds for all files, the multiplicity constraint **some** is used. This forces the restricted relation to contain at least one tuple.

Aggregate functions Lines 22–24 define the value of the size attribute of directories. The size of a directory in the file system is the summation of the sizes of the files which

[¶]The constructs **lone**, **some**, **none** and **one** were introduced in Alloy [Jacozb].

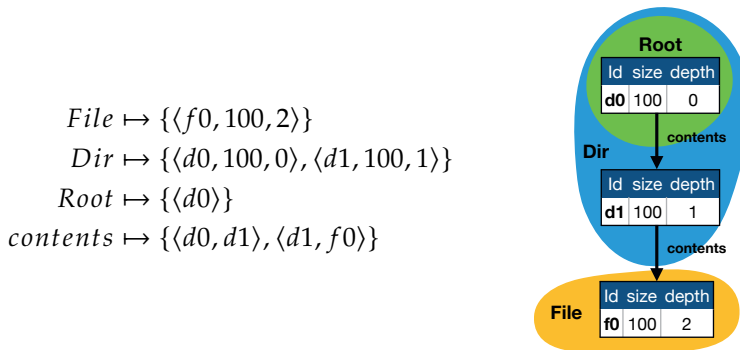


Figure 3.1: The minimal instance of the small file system specification

are (transitively) contained by the directory. On line 23 the `containedFiles` relation of the current directory `d` is defined using the transitive closure of the `contents` relation which is (naturally) joined ($| \times |$) with the current directory `d`. This relation is then used to calculate the size of the directory by using the aggregation function `sum`.

The `sum` function sums up all the values of the `size` attributes in the `containedFiles` relation. The result of applying an aggregation function is yet another relation containing zero or one tuples with one attribute (in this case `totalSize`). Other available aggregation functions include `min`, `max`, `avg` and `count`. `Count` is the only aggregation function that does not need an attribute to perform the aggregation on since it counts the number of tuples in the relation. Note that with the use of the `count` aggregation function all other multiplicity constraints (`some`, `one`, `lone`, `no`) can be expressed.

The remaining constraints describe the value of the `depth` attribute (line 27–29) and enforce the depth of the `Root` directory to be zero (line 32).

Optimization objectives The last line, line 35, defines a single optimization objective. This objective states that we want to optimize on the cardinality of the `File` and `Dir` relations. Only a relation with a single integer attribute can be used as an optimization criterion.

The optimization criteria are so called “soft constraints”. This means that, other than the previously described “hard constraints” (lines 6–31), they do not influence the total number of satisfying instances of a problem. They do influence the order in which the model finder returns solutions. In other words, the instance that is returned first will be the instance that is optimal considering the optimization objectives.

The found solution Figure 3.1 shows the minimal solution of the file system specification. The found solution contains a binding for all the declared relations.

Since the optimization objective stated that we wanted a minimal number of files and directories, it returned a solution that contains only those tuples that were part of our lower bound definition (in the case of `File` and `Root`) and those tuples that were needed to get a consistent model according to the described constraints (in case of the `Dir` and `contents` relations).

Next to that, the model finder returned a binding for all the introduced holes. The definition of the `File` tuple $\langle f0, 2, 100 \rangle$ and the constraints on the data determined the values of the other `depth` and `size` attributes.

3.3 FORMAL DEFINITION OF ALLEALLE

Figure 3.2 shows the abstract syntax of ALLEALLE. We define an ALLEALLE problem as follows. A **problem** P consists of relation definitions $R_1 \dots R_n$ which are bound to a relational variables $r_1 \dots r_n$, formulas $F_1 \dots F_n$ and possibly optimization criteria $O_1 \dots O_n$. A **formula** F is a sentence over an alphabet of the relational variables $r_1 \dots r_n$. A **binding** b is an instance of all the problem's free relational variables to relational constants. A **relational constant** of R is a set of tuples. The binding b is said to be a satisfying instance of P if it conforms to the relation definitions of P and makes all the formulas of P true.

Relations are defined as in the relational model [Cod70; Dat94]. A **relation** R over multiple domains $D_1 \dots D_n$, not necessary distinct, consists of a *header* H and a *body* B . The **header** H consists of a fixed set of attribute names, domain pairs $\{\langle a_1 : D_1 \rangle \dots \langle a_n : D_n \rangle\}$. An **attribute name** is an arbitrary label. A **domain** is a named set of scalar values, all of the same type. Attribute names in a relation are distinct.

A **body** B consists of a set of *tuples* $\{T_1 \dots T_n\}$. Since the body is a true set it means that per definition the tuples in the body must be unique. A **tuple** T consists of a set of *attribute name, value* pairs $\{\langle a_1 : v_1 \rangle \dots \langle a_n : v_n \rangle\}$. For each attribute name and domain pair $\langle a_n : D_n \rangle$ in H there exists an attribute name and value pair $\langle a_n : v_n \rangle$ in T where v_n is drawn from D_n . Relations can be bound to relational variables which are arbitrary labels.

3.3.1 Attribute Domains in ALLEALLE

As stated in the definition above, the attribute domains are named sets of scalar values. Currently ALLEALLE supports the `int` and `id` domains. The scalars of the `int` domain are defined by the underlying SMT solver which is the unbounded set of all integer numbers.

The `id` domain is a bounded domain consisting of arbitrary chosen labels (atoms). Like mentioned earlier, it contains exactly those atoms that are defined in a specification.

See Appendix B.2 for the concrete syntax definition of ALLEALLE.

Please note that the existence of this domain is not strictly essential (since it could be modeled using `int` values) but it allows for a convenient way to model different dependencies between relations (associations, containment, specialization, etc.).

3.3.2 Semantics

Figure 3.3 shows the semantics of ALLEALLE. We do not include the semantics of the optimization objectives since they are orthogonal to the semantics of formulas and expressions, and defined in terms of the underlying SMT solver, *vZ* [BPF15]. The meaning of an ALLEALLE problem is defined by four functions, P , R , F and E , which can be recursively applied. The function R accepts a relation declaration and binding and returns whether the header of the binding is equal to the header of the declaration and whether the lower bound of the declaration is a subset of the body of the binding which in turn must be a subset of the upper bounds of the declaration. The function F accepts an ALLEALLE formula and a binding and returns whether the binding satisfies the formula. The function E accepts an ALLEALLE expression and a binding and returns a relational constant. The function P acts as the starting point and accepts a `Problem` and a binding, and calls the R and F functions.

The semantics of the rename, project, restrict, and aggregate operators on relations are defined in the standard way [Cod70]. The same goes for union (\cup), intersection (\cap), difference (\setminus) and cartesian product (\times). Union, intersection and difference can only be applied on union compatible relations (see 3.2.1). Cartesian product requires two relations with disjoint headers. Applying the cartesian product on two relations with respectively n and m sized tuples *flattens* both relations into a new relation of $n + m$ sized tuples.

The semantics of ALLEALLE shown in Figure 3.3 defines the meaning of an ALLEALLE problem given an assignment of relation constants to all relational variables in the binding b . In order to find solutions rather than check their truth value, however, ALLEALLE problems are translated to SMT formulas.

```

problem ::=  $\overline{\text{relDecl}} \overline{\text{alleForm}} \overline{\text{objective}}$ 
relDecl ::=  $x ( \text{relHeader} ) \text{relBody}$ 
relHeader ::=  $\{x : \text{domain}\}$ 
relBody ::=  $= \text{bound} \mid \leq \text{bound} \mid \geq \text{bound} \mid \leq \text{bound}$ 
            $\text{bound} ::= \overline{\text{tupleDecl}}$ 
tupleDecl ::=  $\overline{\langle \text{value} \rangle}$ 
           value ::=  $x \mid n \mid ?$ 
           domain ::=  $\text{id} \mid \text{int}$ 

alleForm ::= not alleForm                                negation
           | no alleExpr                                empty
           | lone alleExpr                              at most one
           | one alleExpr                              exactly one
           | some alleExpr                              at least one
           | alleExpr in alleExpr                      subset
           | alleExpr = alleExpr                        equal
           | alleForm || alleForm                      disjunction
           | alleForm && alleForm                      conjunction
           | alleForm => alleForm                      implication
           | alleForm <=> alleForm                    equality
           | forall  $\overline{x : \text{alleExpr}}$  | alleForm        universal
           | exists  $\overline{x : \text{alleExpr}}$  | alleForm        existential
           | let  $\overline{x = \text{alleExpr}}$  | alleForm          let

alleExpr ::=  $x$ 
           | alleExpr  $\overline{[x \text{ as } \bar{x}]}$                 renaming
           | alleExpr  $\overline{[\bar{x}]}$                     projection
           | alleExpr where condition                restriction
           |  $\sim$ alleExpr                                trans. closure
           |  $*$ alleExpr                                refl. trans. clos.
           | alleExpr  $\overline{[\text{aggFunc}]}$                 aggregate
           | alleExpr + alleExpr                    union
           | alleExpr & alleExpr                    intersection
           | alleExpr - alleExpr                    difference
           | alleExpr × alleExpr                    product
           | alleExpr |×| alleExpr                    natural join

condition ::= not condition
           | condition && condition
           | condition || condition
           | conditionExpr (< | ≤ | > | ≥ | =) conditionExpr

conditionExpr ::=  $x \mid n \mid \text{conditionExpr} \mid - \text{conditionExpr}$ 
           |  $\text{conditionExpr} ( + \mid - \mid * \mid / \mid \% ) \text{conditionExpr}$ 

aggFunc ::= count() | sum(x) | min(x) | max(x) | avg(x)    agg. func.

objective ::= maximize alleExpr | minimize alleExpr        obj. crit.

```

Figure 3.2: Abstract Syntax of ALLEALLE

P	$: \text{problem} \rightarrow \text{binding} \rightarrow \text{boolean}$
R	$: \text{relDecl} \rightarrow \text{binding} \rightarrow \text{boolean}$
F	$: \text{alleForm} \rightarrow \text{binding} \rightarrow \text{boolean}$
E	$: \text{expr} \rightarrow \text{binding} \rightarrow \text{constant}$
binding	$: \text{var} \rightarrow \text{constant}$
$P[r_1 \dots r_n f_1 \dots f_m]b$	$= R[r_1]b \wedge \dots \wedge R[r_n]b \wedge F[f_1]b \wedge \dots \wedge F[f_m]b$
$R[x (h) [l, u]]b$	$= h = b[x]_{\text{header}} \wedge l \subseteq b[x]_{\text{body}} \subseteq u$
$F[\text{not } f]b$	$= \neg F[f]b$
$F[\text{no } r]b$	$= E[r]b = 0$
$F[\text{lone } r]b$	$= E[r]b \leq 1$
$F[\text{one } r]b$	$= E[r]b = 1$
$F[\text{some } r]b$	$= E[r]b > 0$
$F[r \text{ in } s]b$	$= E[r]b \subseteq E[s]b$
$F[r = s]b$	$= E[r]b \subseteq E[s]b \wedge E[s]b \subseteq E[r]b$
$F[f \parallel g]b$	$= F[f]b \vee F[g]b$
$F[f \&\& g]b$	$= F[f]b \wedge F[g]b$
$F[f \Rightarrow g]b$	$= \neg F[f]b \vee F[g]b$
$F[f \Leftrightarrow g]b$	$= F[f]b \iff F[g]b$
$F[\text{forall } v_1 : r_1 \dots v_n : r_n \mid f]b$	$= \bigwedge_{t \in E[e_1]b} (F[\text{forall } v_2 : e_2 \dots v_n : r_n \mid f](b \oplus v_1 \mapsto \{t\}))$
$F[\text{exists } v_1 : r_1 \dots v_n : r_n \mid f]b$	$= \bigvee_{t \in E[r_1]b} (F[\text{exists } v_2 : r_2 \dots v_n : r_n \mid f](b \oplus v_1 \mapsto \{t\}))$
$F[\text{let } v_1 : r_1 \dots v_n : r_n \mid f]b$	$= F[\text{let } v_2 : r_2 \dots v_n : r_n \mid f](b \oplus v_1 \mapsto E[r_1]b)$
$E[x]b$	$= b[x]$
$E[r[a_1 \text{ as } aa_1, \dots a_n \text{ as } aa_n]]b$	$= \rho_{(aa_1/a_1 \dots aa_n/a_n)} E[r]b$
$E[r[a_1 \dots a_n]]b$	$= \prod_{(a_1 \dots a_n)} E[r]b$
$E[r \text{ where } c]b$	$= \sigma_c E[r]b$
$E[\neg r]b$	$= \text{let } m \leftarrow E[r]b \text{ in } \langle m_{\text{header}}, \{(x : id_x, y : id_y) \mid \exists id_1 \dots id_n (x : id_x, y : id_1), (x : id_1, y : id_2) \dots (x : id_n, y : id_y) \in m_{\text{body}}\} \rangle$
$E[\neg r]b$	$= E[\neg r]b \cup \mathbb{I}$
$E[r[f()]]b$	$= f()E[r]b$ (where f is count)
$E[r[f(a)]]b$	$= f(a)E[r]b$ (where f is sum, avg, min or max)
$E[r + s]b$	$= E[r]b \cup E[s]b$
$E[r \& s]b$	$= E[r]b \cap E[s]b$
$E[r - s]b$	$= E[r]b \setminus E[s]b$
$E[r \times s]b$	$= E[r]b \times E[s]b$
$E[r \mid \times \mid s]b$	$= E[r]b \bowtie E[s]b$

Figure 3.3: Semantics of ALLEALLE. Variables f and g range over formulas, r_n and s over expressions. The \oplus operator updates bindings. \mathbb{I} represents the binary identity relation on all values in the **id** domain. \cup , \cap , \setminus and \times have their standard relational algebra semantics.

3.4 TRANSLATING ALLEALLE TO SMT

Specifications are translated to SMT constraints. Figure 3.4 describes the definition of the resulting formula (FORM) that the translation algorithm produces. Our prototype of ALLEALLE translates ALLEALLE problems to the standard SMT-LIB format, which is supported by multiple SMT solvers [BST10]. As a result, ALLEALLE can potentially be used in combination with different SMT solvers as backends.**

Apart from the optimization criteria, the translation consists of flattening ALLEALLE problems to a single SMT formula within the logic fragment of quantifier-free non-linear integer arithmetic (QF-NIA). This means that the final SMT formula is a large, but flat formula made up of negation, conjunction, disjunction, integer arithmetic, (in)equalities and if-then-else constructs, as shown in Figure 3.4.

Before we go into the details of the translation rules we will give an example of how translation unfolds for a small problem.

3.4.1 Translation Example

Assume we have a relation `Person` with two attributes `pId` and `age` which is defined as follows:

```
Person (pId: id, age: int) <= {<p1,17>,<p2,?>}
```

This relation has an upper bound containing two tuples. The lower bound is omitted and thus empty (i.e., the empty set). Consequently, a satisfying instance may hold zero, one or two tuples in the `Person` relation. The first tuple, `<p1,17>`, assigns the value 17 to the `age` attribute. This means that if this tuple is present the value of `age` must be 17. In the second tuple, `<p2,?>`, the value of the `age` is left open meaning that the value is left to the underlying solver.

Next we define the following constraint:

```
some Person where age >= 18
```

This constraint states that there must be at least one person who is an adult. Or more precisely, there needs to be at least one tuple in the `Person` relation where the value of the `age` attribute is equal to or greater than 18. Please note that the first tuple in the relation, `<p1,17>`, can never satisfy this constraint since the value of its `age` attribute will always be 17.

**Currently *optimization criteria* are not supported by all SMT solvers. At least Z3 [BPF15] and MathSAT5 [ST15] have built-in support.

$$\begin{aligned}
form & ::= \top \mid \perp \mid x \mid \neg form \mid form \wedge form \mid form \vee form \mid \\
& \quad expr (< \mid \leq \mid = \mid \geq \mid >) expr \\
expr & ::= literal \mid x \mid expr + expr \mid expr - expr \mid expr * expr \mid \\
& \quad expr / expr \mid expr \% expr \mid form ? expr : expr
\end{aligned}$$

Figure 3.4: Definition of *form* and *expr*.

As a first step in the translation an environment ρ is constructed, mapping relation names (e.g., `Person`) to an internal relation representation. In the example the created environment ρ is as follows:

$$\rho = \left(\text{Person} \mapsto \begin{array}{cc|cc} \text{pId} & \text{age} & \text{exists} & \text{attCons} \\ \hline p1 & 17 & b_0 & \top \\ p2 & i_0 & b_1 & \top \end{array} \right)$$

The internal representation of the `Person` relation consists of a table, with columns for the declared attributes `pId` and `age`, and two additional columns, `EXISTS` and `ATTCONS`. The `pId` attribute contains the values `p1` and `p2` both drawn from the `id` domain. For the first tuple the `age` attribute contains the constant `17`. This is a consequence of the given relation definition where the `age` attribute for this tuple was assigned `17`. For the second tuple the `age` attribute contains an integer variable `i0`. Since in the relation definition this value was left open it is converted to a fresh integer variable.

The `EXISTS` column encodes whether the tuple should be present in a satisfying instance or not. In this case the value of the `EXISTS` column for both tuples contains a fresh boolean variable, `b0` and `b1` respectively. This is due to the fact that both tuples are part of the upper bound of the relation but not of the lower bound (since the lower bound of this relation is the empty set). For each satisfying instance the solver will assign truth values to these variables. For instance, if `b0 = \top` and `b1 = \perp` it means that the tuple `<p1, 17>` is in the `Person` relation but `<p2, ?>` is not. The `ATTCONS` column encodes constraints formulated on the attribute values. Initially the `ATTCONS` attributes have `\top` assigned.

The next step in the translation is the translation of the constraints. The translation of constraints consists of the recursive application of two translation functions T_F for the translation of `ALLEALLE` formulas and T_E for the translation of `ALLEALLE` expressions. Their full definitions are shown in Figures 3.8 and 3.9. The full translation tree for this example would look like:

$$\begin{array}{c}
T_F[\text{some Person where age} \geq 18]\rho \\
| \\
T_E[\text{Person where age} \geq 18]\rho \\
| \\
T_E[\text{Person}]\rho
\end{array}$$

We describe the translation of the example in a bottom-up fashion. The first expression that is translated is the lookup of the `Person` relation from the environment ρ :

$$T_E[\text{Person}]\rho = \rho(\text{Person}) = \begin{array}{cc|cc} \text{pId} & \text{age} & \text{exists} & \text{attCons} \\ \hline p1 & 17 & b_0 & \top \\ p2 & i_0 & b_1 & \top \end{array}$$

As shown above, the result of the translation function T_E is another relation. Now the outer `where` expression is translated as follows:

$$T_E[\text{Person where age} \geq 18]\rho = \begin{array}{cc|cc} \text{pId} & \text{age} & \text{exists} & \text{attCons} \\ \hline p1 & 17 & b_0 & \perp \\ p2 & i_0 & b_1 & i_0 \geq 18 \end{array}$$

The restriction expression (`age` \geq 18) forces additional constraints on the `age` attribute. In case of the first tuple the `age` attribute contains the constant 17. Since 17 is less than 18, the value \perp is assigned to the `ATTCONS` attribute. For the second tuple the `age` attribute was left open which resulted in the introduction of the i_0 variable. For this tuple the constraint $i_0 \geq 18$ is added to the `ATTCONS` column.

The last step is the translation of the outer formula:

$T_F[\text{some Person where age} \geq 18]\rho$. The T_F function flattens the relation into a flat SMT formula. The translation of the `some` operator gives the following result:

$$\begin{aligned}
& T_F[\text{some Person where age} \geq 18]\rho \\
&= \bigvee \left(\begin{array}{cc|cc} \text{pId} & \text{age} & \text{exists} & \text{attCons} \\ \hline p1 & 17 & b_0 & \perp \\ p2 & i_0 & b_1 & i_0 \geq 18 \end{array} \right) \\
&= (b_0 \wedge \perp) \vee (b_1 \wedge i_0 \geq 18) \\
&= b_1 \wedge i_0 \geq 18
\end{aligned}$$

The `some` formula is satisfied if at least one tuple in the relation exists. This is accomplished by translating it to a disjunction of the conjoined `EXISTS` and `ATTCONS` columns of the tuples in the relation (i.e., this is depicted by the big vee notation in the above translation).

$$\begin{aligned}
rel & ::= \langle \overline{x: domain_{header}}, \overline{tuple_{body}} \rangle \\
tuple & ::= \langle \overline{(x_{name}: cell_{value})_{attributes}}, form_{exists}, form_{attCons} \rangle \\
cell & ::= atom \mid expr \\
domain & ::= ID \mid INT
\end{aligned}$$

Figure 3.5: Definition of *rel* as used in the translation. *expr* and *form* are defined in Figure 3.4.

As can be seen, the translation of this formula results in the SMT formula $b_1 \wedge i_0 \geq 18$. This means that an instance of this problem is satisfying iff it contains the second tuple (i.e., b_1 must be true) and the value of its age attribute is greater than or equal to 18. The presence or absence of the first tuple does not change the validity of the resulting instance since it assigned age value of 17 never conforms to the formulated constraint. This means that a satisfying instance can either contain or not contain the first tuple as long as the second tuple is present.

3.4.2 The Algorithm

The translation of an ALLEALLE specification starts with the translation of a problem using the function T_P :

$$\begin{aligned}
T_P &: problem \rightarrow form \\
T_P[r_1(h_1)b_1 \dots r_n(h_n)b_n \ f_1 \dots f_m] &= \bigwedge_{i=1}^m T_F[f_i]\rho \\
\text{where } \rho &= \bigcup_{j=1}^n (r_j \mapsto T_R[(h_j) \ b_j])
\end{aligned}$$

This function translates the constraints $f_1 \dots f_m$ of the problem to formulas (of type *form*, see Figure 3.4). The environment is populated using the T_R function which converts relation declarations to the internal tabular representation. The T_P function returns a conjunction of all the translated constraints.

The Relation data structure Central to the translation is the internal relation data structure, shown in Figure 3.5, which in this paper we visualize using the tabular notation introduced above. A relation *rel* consists of a header and a body. The header is defined as a mapping from attribute names to domains. The body is a set of tuples. Each tuple in the body contains the declared `ATTRIBUTES` and two additional columns, `EXISTS` and `ATTCONS`. When we refer to *tuple* we refer to the combination of the `ATTRIBUTES` and `EXISTS` and `ATTCONS` columns. In the translation rules we will use the subscripts r_{header} and r_{body} for the header and body of a relation r and $t_{attributes}$, t_{exists} and $t_{attCons}$ for the `ATTRIBUTES`, `EXISTS` and `ATTCONS` columns of a tuple t to refer to the specific parts of a relation or tuple (see Figure 3.5).

$$\begin{aligned}
T_R & : relHeader \rightarrow relBody \rightarrow rel \\
T_R[(h) = b] & = add(\langle h, \emptyset \rangle, b, \lambda t. \langle convert(t), \top, \top \rangle) \\
T_R[(h) \leq ub] & = add(\langle h, \emptyset \rangle, ub, \lambda t. \langle convert(t), x, \top \rangle) \\
T_R[(h) \geq lb \leq ub] & = add(\langle h, \emptyset \rangle, ub, \lambda t. \langle convert(t), exists(t, lb), \top \rangle) \\
\\
add & : rel \rightarrow \overline{tupleDecl} \rightarrow (tupleDecl \rightarrow tuple) \rightarrow rel \\
add[r, b, f] & = \text{if } b = \emptyset \text{ then } r \text{ else let } t \in b \text{ in } add(addDistinct(r, f(t)), b \setminus t) \\
\\
convert & : tupleDecl \rightarrow \overline{x : cell} \\
convert[t] & = \langle atr_{name} : \begin{cases} id & \text{when } atr = id \\ i & \text{when } atr = hole \text{ (} i \text{ as fresh int var)} \\ constant & \text{when } atr = constant \end{cases} \mid atr \in t \rangle \\
\\
exists & : tupleDecl \rightarrow bound \rightarrow form \\
exists[t, lb] & = \begin{cases} \top & \text{when } t \in lb \\ x & \text{otherwise (with } x \text{ as fresh bool var)} \end{cases} \\
\\
addDistinct & : rel \rightarrow tuple \rightarrow rel
\end{aligned}$$

Figure 3.6: Definition and construction of relations. The definition of the *addDistinct* function is included in Appendix C.1. *relHeader*, *relBody*, *bound*, *tupleDecl*, *id* and *hole* are declared in Figure 3.2. The other definitions are given in Figure 3.4 and Figure 3.5.

	oId	depth	size	exists	attCons
File \mapsto	<i>f0</i>	2	100	\top	\top
	<i>f1</i>	<i>i₀</i>	<i>i₁</i>	<i>b₀</i>	\top
	<i>f2</i>	<i>i₂</i>	<i>i₃</i>	<i>b₁</i>	\top

Figure 3.7: The visual representation of File REL after its construction based on the relation declaration: File (oId:**id**,depth:**int**,size:**int**) \geq {<f0,2,100>} \leq {<f0,2,100>,<f1,?,?>,<f2,?,?>}.

Constructing relations

A REL can be constructed in three different ways depending on how it is declared. Figure 3.6 shows the definition of the construction function T_R . This function translates the relation definition (i.e., the header and lower and upper bounds) to a REL. Which construction function is used depends on how the bounds are declared and influences the value of the tuple's EXISTS field. This value depends on whether the tuple declaration is part of the lower and upper bound or only of the upper

bound. For instance, in the example of the small file system (Figure 3.1) the `File` relation is declared with both a lower and an upper bound (e.g., $\text{>= } \{ \langle f0, 2, 100 \rangle \} \text{ <= } \{ \langle f0, 2, 100 \rangle, \langle f1, ? \rangle, \langle f2, ? \rangle \}$).

On construction of the `REL` the value \top is assigned to the `EXISTS` field of the tuple $\langle f0, 2, 100 \rangle$ since there cannot be a satisfying instance without this tuple present. It is a different case for the tuples $\langle f1, ? \rangle$ and $\langle f2, ? \rangle$. Since they are only part of the relation's upper bound there may be satisfying instances where these tuples (or one of these tuples) are not present. To encode this, fresh boolean variables are assigned to the `EXISTS` fields of both tuples. It is then up to the underlying SMT solver to find a satisfying assignment for these boolean variables. The full encoding of the `File` relation is shown in Figure 3.7 (with the `EXISTS` column highlighted). The encoding is adapted from the relational model finder `KODKOD`, with the difference that it is encoded in our relation data structure instead of a boolean matrix as used by `KODKOD` [Tj07a].

Ensuring tuple distinctness The `ATTCONS` column holds the constraints that were added on the tuple's scalar attributes. On construction this column will be populated with \top for most tuples. The only exception to this is when the added tuples in the relation are potentially non-distinct. Consider for instance the (valid) case that the `File` relation would have an upper bound of two possible tuples: $\langle f1, 1, 10 \rangle, \langle f1, ? \rangle$. Since they both share the same `oid` value (namely `f1`) these tuples could potentially overlap if the `depth` and `size` attributes of the second tuple would evaluate to 1 and 10 respectively.

Since the relational model dictates true set semantics for its relational bodies, the translation must enforce that all tuples in the relation are distinct, or collapsed into each other. To enforce this the translation algorithm adds a constraint to the `ATTCONS` field of the second tuple that forces the value of either the `depth` or `size` attribute to be different, if the first tuple exists in the relation.

This distinctness rule is added on construction of the relations by applying the `addDistinct` function (included in Appendix C.1). This function checks whether tuples can potentially overlap and adds the necessary constraints to the `ATTCONS` field. In the case of the example given in this section the constructed relation would be as follows:

	oid	depth	size	exists	attCons
<code>File</code> \mapsto	<code>f1</code>	1	10	b_0	\top
	<code>f1</code>	i_0	i_1	b_1	$\neg b_0 \vee \neg(i_0 = 1 \wedge i_1 = 10)$

Translating ALLEALLE constraints

The entry for translating ALLEALLE formulas to SMT formulas is the T_F function, shown in Figure 3.8. To translate an ALLEALLE expression the T_F function calls the T_E function which is defined in Figure 3.9.

The T_E function translates the expression and returns a new REL. The T_F function flattens the translated RELS into SMT formulas. These in turn get conjoined by the T_P function introduced earlier.

Tuple equality constraints When translating the *subset* formula and the *union*, *intersection* and *difference* expressions we again have to account for possible overlapping tuples described earlier. The difference being that in the case of translating the above rules we need to *enforce* equality instead of preventing it. In this case, the constraints that need to be added to the ATTCONS field are constraints that force the value of the attributes to be the same; this is done using the helper functions *canOverlap* and *attEquals*, which are included in Appendix C.1.

As an example consider the following case. Suppose we have the following specification:

```
Shape (sid: id, size: int) <= {<s1, ?>}
Square (sid: id, size: int) <= {<s1, ?>}
```

```
Square in Shape
```

We define two relations, Shape and Square. Both relations have an `sid` field of type `id` and a `size` field of type `int`. The `sid` fields contain the same `id` literal and both `size` attributes have been left open. Translating the constraint Square `in` Shape would yield the following result:

$$\begin{aligned}
 T_E[\text{Shape}]\rho &= \frac{\text{sid} \quad \text{size}}{s_1 \quad i_0} \mid \frac{\text{exists} \quad \text{attCons}}{b_0 \quad \top} \\
 T_E[\text{Square}]\rho &= \frac{\text{sid} \quad \text{size}}{s_1 \quad i_1} \mid \frac{\text{exists} \quad \text{attCons}}{b_1 \quad \top} \\
 T_F[\text{Square } \mathbf{in} \text{ Shape}]\rho &= \neg b_1 \vee (b_0 \wedge i_0 = i_1)
 \end{aligned}$$

The outcome is that a Square can only be a Shape if either the Square relation is empty (by enforcing that $b_1 = \perp$) or the value of the `size` attributes of both relations is equal (by enforcing that $i_0 = i_1$). Otherwise the tuple in Square would not overlap with the tuple in Shape and thus would not be in the *subset* relation.

Translation of the projection expression Projection can reduce the numbers of the tuples in the relation by truncating it. This again can potentially cause tuple

$env : identifier \rightarrow rel$

$T_F : alleForm \rightarrow env \rightarrow form$

$T_F[\mathbf{not} f]\rho = \neg T_F[f]\rho$

$T_F[\mathbf{no} r]\rho = \neg T_F[\mathbf{some} r]\rho$

$T_F[\mathbf{lone} r]\rho = T_F[\mathbf{no} r]\rho \vee T_F[\mathbf{one} r]\rho$

$T_F[\mathbf{one} r]\rho = \text{let } m \leftarrow T_E[r]\rho \text{ in } \bigvee_{t \in m_{body}} \left(\text{tg}(t) \wedge \left(\bigwedge_{t' \in m_{body}, t' \neq t} (\neg \text{tg}(t')) \right) \right)$

$T_F[\mathbf{some} r]\rho = \text{let } m \leftarrow T_E[p]\rho \text{ in } \bigvee_{t \in m_{body}} (\text{tg}(t))$

$T_F[r \mathbf{in} s]\rho = \text{let } m \leftarrow T_E[r]\rho, \text{ let } n \leftarrow T_E[s]\rho$

$\left(\bigwedge_{t \in m_{body}, u \in n_{body}, \text{canOverlap}(t, u)} (\neg t \text{g}(t) \vee (t \text{g}(u) \wedge \text{attEqual}(t, u))) \right) \wedge$
 $\left(\bigwedge_{t \in m_{body}, t \notin n_{body}} \neg t \text{g}(t) \right)$

$T_F[r = s]\rho = T_F[r \mathbf{in} s]\rho \wedge T_F[s \mathbf{in} r]\rho$

$T_F[f \parallel g]\rho = T_F[f]\rho \vee T_F[g]\rho$

$T_F[f \ \&\& \ g]\rho = T_F[f]\rho \wedge T_F[g]\rho$

$T_F[f \Rightarrow g]\rho = \neg T_F[f]\rho \vee T_F[g]\rho$

$T_F[f \Leftrightarrow g]\rho = T_F[f \Rightarrow g]\rho \wedge T_F[g \Rightarrow f]\rho$

$T_F[\mathbf{forall} v_1 : ex_1 \dots v_n : ex_n \mid f]\rho = \text{let } m \leftarrow T_E[ex_1]\rho \text{ in}$

$\bigwedge_{t \in m_{body}} (\neg t \text{g}(t) \vee T_F[\mathbf{forall} v_2 : ex_2 \dots v_n : ex_n \mid f]\rho[v_1 \mapsto \text{sing}(m_{header}, t)])$

$T_F[\mathbf{exists} v_1 : ex_1 \dots v_n : ex_n \mid f]\rho = \text{let } m \leftarrow T_E[ex_1]\rho \text{ in}$

$\bigvee_{t \in m_{body}} (t \text{g}(t) \wedge T_F[\mathbf{exists} v_2 : ex_2 \dots v_n : ex_n \mid f]\rho[v_1 \mapsto \text{sing}(m_{header}, t)])$

$T_F[\mathbf{let} v_1 : ex_1 \dots v_n : ex_n \mid f]\rho = T_F[\mathbf{let} v_2 : ex_2 \dots v_n : ex_n \mid f](\rho[v_1 \mapsto T_E[ex_1]\rho])$

$\text{tg} : tuple \rightarrow form$

$\text{tg}[t] = t_{exists} \wedge t_{attCons}$

$\text{sing} : \overline{x : domain} \rightarrow tuple \rightarrow rel$

$\text{sing}[h, t] = \langle h, \{ \langle t_{attributes}, \top, t_{attCons} \rangle \} \rangle$

Figure 3.8: Translation rules for ALLEALLE formulas. r and s are *alleExpr*, f and g are *alleForm*. Definitions of *canOverlap* and *attEqual* are included in Appendix C.1. *tg* is short for *together* and *sing* is short for *singleton* meaning a relation with only one tuple.

overlap. Consider for instance the `Person` relation introduced in the translation example (Section 3.4.1). This relation has two attributes, `pId` and `age` and was defined as follows:

`Person (pId: id, age: int) <= {<p1, 17>, <p2, ?>}`.

resulting in the following internal representation:

$$\begin{aligned}
T_E &: \text{alleExpr} \rightarrow \text{env} \rightarrow \text{rel} \\
T_E[v] &= \rho(v) \\
T_E[r[a_1 \text{ as } a'_1 \dots a_n \text{ as } a'_n]] &= \text{let } m \leftarrow T_E[r]\rho \text{ in} \\
&\quad T_E[\langle m_h[a_1/a'_1], m_b[a_1/a'_1][a_2 \text{ as } a'_2 \dots a_n \text{ as } a'_n] \rangle]\rho \\
T_E[r[a_1 \dots a_n]] &= \text{let } m \leftarrow T_E[r]\rho \text{ in} \\
&\quad \langle (a: m_h[a] \mid a \in a_1 \dots a_n), \text{merge}(m_b, a_1 \dots a_n) \rangle \\
T_E[r \text{ where } c] &= \text{let } m \leftarrow T_E[r]\rho \text{ in } \langle m_h, \{ \langle t_a, t_e, t_a \wedge T_C[c]t \rangle \mid t \in m_b \} \rangle \\
T_E[\wedge r] &= \text{let } m \leftarrow T_E[r]\rho \text{ in} \\
&\quad \text{let } \text{sqr} \leftarrow \lambda r. i. \text{ if } i >= |m_{\text{body}}| \text{ then } r \text{ else } \text{joinOnce}(r, i * 2) \text{ in } \text{sqr}(m, 1) \\
T_E[*r] &= T_E[\text{idem} + \wedge r]\rho \\
\\
T_E[r[f(a) \text{ as } x]] &= \text{let } m \leftarrow T_E[r]\rho \text{ in (where } f \text{ is } \text{count, sum, avg}) \\
&\quad \langle (x: \text{int}), \langle (x: i), \top, i = T_{Ag}[f(a)]m_b \rangle \rangle \\
T_E[r[f(a) \text{ as } x]] &= \text{let } m \leftarrow T_E[r]\rho \text{ in (where } f \text{ is } \text{min, max}) \\
&\quad \langle (x: \text{int}), \langle (x: i), \text{if } |m| > 0 \text{ then } \top \text{ else } \perp, i = T_{Ag}[f(a)]m_b \rangle \rangle \\
\\
T_E[r+s] &= \text{let } m \leftarrow T_E[r]\rho, \text{ let } n \leftarrow T_E[s]\rho \text{ in} \\
&\quad \langle m_h, \{ \langle t_a, t_e \vee u_e, t_c \wedge u_c \wedge \text{attEqual}(t, u) \rangle \mid t \in m_b, u \in n_b, \text{canOverlap}(t, u) \\
&\quad \cup (m_b \setminus n_b) \cup (n_b \setminus m_b) \} \rangle \\
T_E[r \& s] &= \text{let } m \leftarrow T_E[r]\rho, \text{ let } n \leftarrow T_E[s]\rho \text{ in} \\
&\quad \langle m_h, \{ \langle t_a, t_e \wedge u_e, t_c \wedge u_c \wedge \text{attEqual}(t, u) \rangle \mid t \in m_b, u \in n_b, \text{canOverlap}(t, u) \} \rangle \\
T_E[p \cdot q] &= \text{let } m \leftarrow T_E[p]\rho, \text{ let } n \leftarrow T_E[q]\rho \text{ in} \\
&\quad \langle m_h, \{ \langle t_a, t_e \wedge \neg u_e, t_c \wedge \text{attEqual}(t, u) \rangle \mid t \in m_b, u \in n_b, \text{canOverlap}(t, u) \\
&\quad \cup (m_b \setminus n_b) \} \rangle \\
T_E[r \times s] &= \text{let } m \leftarrow T_E[r]\rho, \text{ let } n \leftarrow T_E[s]\rho \text{ in} \\
&\quad \langle m_h \cup n_h, \{ \langle t_a \cup u_a, t_e \wedge u_e, t_c \wedge u_c \rangle \mid t \in m_b, u \in n_b, t_a \cup u_a = \emptyset \} \rangle \\
T_E[p \mid \times \mid q] &= \text{let } m \leftarrow T_E[p]\rho, \text{ let } n \leftarrow T_E[q]\rho \text{ in} \\
&\quad \langle m_h \cup n_h, \{ \langle t_a \cup u_a, t_e \wedge u_e, t_c \wedge u_c \rangle \mid t \in m_b, u \in n_b, t_a \cup u_a \neq \emptyset \} \rangle
\end{aligned}$$

$$\text{merge: } \overline{\text{tuple}} \rightarrow \bar{x} \rightarrow \overline{\text{tuple}}$$

$$\text{joinOnce: } \text{rel} \rightarrow x \rightarrow y \rightarrow \text{rel}$$

Figure 3.9: Translation rules for ALLEALLE expressions. For abbreviation purposes the notation r_h means r_{header} , r_b means r_{body} , t_a means $t_{\text{attributes}}$, t_e means t_{exists} and t_c means t_{attCons} . r and s are *alleExpr*, c is a *condition*. Definitions of *canOverlap* and *attEqual* are included in Appendix C.1. The definition of *merge* and *joinOnce* is not given but sketched in the text. **iden** resolves to the binary identity relation on all values in the **id** domain.

	pId	age	exists	attCons
Person \mapsto	$p1$	17	b_0	\top
	$p2$	i_0	b_1	\top

$$\begin{aligned}
T_C & : tuple \rightarrow form \\
T_C[!c]t & = \neg T_C[c]t \\
T_C[c_1 \ \&\& \ c_2]t & = T_C[c_1]t \wedge T_C[c_2]t \\
T_C[c_1 \ || \ c_2]t & = T_C[c_1]t \vee T_C[c_2]t \\
T_C[e_1 \ \odot \ e_2]t & = T_{C_e}[e_1]t \ \odot \ T_{C_e}[e_2]t
\end{aligned}$$

$$\begin{aligned}
T_{C_e} & : tuple \rightarrow expr \\
T_{C_e}[x]t & = t_{attributes}[x] \\
T_{C_e}[n]t & = n \\
T_{C_e}[- \ e]t & = \neg T_{C_e}[e]t \\
T_{C_e}[|e|]t & = \text{let } i \leftarrow T_{C_e}[e]t \text{ in } i < 0? -i : i \\
T_{C_e}[e_1 \ \oplus \ e_2]t & = T_{C_e}[e_1]t \ \oplus \ T_{C_e}[e_2]t
\end{aligned}$$

Figure 3.10: Translation rules for the restriction conditions. \odot depicts the different equality operators ($<$, \leq , $=$, \geq , $>$), \oplus depicts the arithmetic operators ($+$, $-$, $*$, $/$, $\%$).

If we would project the `age` attribute both tuples could potentially collapse into each other. This would be the case if both tuples exist and the value of the `age` attribute of the second tuple would also be 17. To prevent this the `merge` function adds extra constraints to the tuples reusing the `addDistinct` function (see Appendix C.1) enforcing that in all possible evaluations of its variables the result would be a relation with distinct tuples. We would end up with a relation with the following values and constraints:

	age	exists	attCons
Person \mapsto	17	b_0	\top
	i_0	b_1	$\neg b_0 \vee \neg(i_0 = 17)$

Translation of the aggregation expression Aggregation results in a new relation containing zero or one tuple with a single attribute. The value of the `ATTCONS` field of this tuple contains the unfolded aggregation expression. The translation of the different rules are shown in Figure 3.11.

In case of the application of the `count`, `sum` and `avg` aggregation the resulting relation will always contain a single tuple, even if the aggregated relation was empty. The intuition behind this is that even if the aggregated relation is empty its cardinality is zero and the sum of one of its attributes will also result in zero. The resulting relation after applying the `min` and `max` aggregation could be empty since calculating the `max` or `min` of an empty relation is undefined.

Translation of the transitive closure expression Transitive closure is only defined for binary relations containing two `id` attributes. In theory it is possible to define

$$\begin{aligned}
T_{Ag} &: aggFunc \rightarrow \overline{tuple} \rightarrow form \\
T_{Ag}[\mathbf{count}()]b &= \text{let cnt} \leftarrow \lambda b'.term. \text{ if } b' = \emptyset \text{ then } term \text{ else} \\
&\quad \text{let } t \in b' \text{ in } term + \text{cnt}(b' \setminus t, \text{tg}(t) ? 1 : 0) \text{ in } \text{cnt}(b, 0) \\
T_{Ag}[\mathbf{sum}(a)]b &= \text{let sum} \leftarrow \lambda b'.term. \text{ if } b' = \emptyset \text{ then } term \text{ else} \\
&\quad \text{let } t \in b' \text{ in } term + \text{sum}(b' \setminus t, \text{tg}(t) ? t_{attributes}[a] : 0) \text{ in } \text{sum}(b, 0) \\
T_{Ag}[\mathbf{avg}(a)]b &= T_{Ag}[\mathbf{sum}(a)]b \div T_{Ag}[\mathbf{count}()]b \\
T_{Ag}[\mathbf{min}(a)]b &= \text{let min} \leftarrow \lambda b'.term. \text{ if } b' = \emptyset \text{ then } term \text{ else} \\
&\quad \text{let } t \in b' \text{ in } t_{attributes}[a] < term \wedge \text{tg}(t) ? \\
&\quad \quad \text{min}(b' \setminus t, t_{attributes}[a]) : \text{min}(b' \setminus t, term) \text{ in } \text{min}(b, \text{findFirst}(a, b)) \\
T_{Ag}[\mathbf{max}(a)]b &= \text{let max} \leftarrow \lambda b'.term. \text{ if } b' = \emptyset \text{ then } term \text{ else} \\
&\quad \text{let } t \in b' \text{ in } t_{attributes}[a] > term \wedge \text{tg}(t) ? \\
&\quad \quad \text{max}(b' \setminus t, t_{attributes}[a]) : \text{max}(b' \setminus t, term) \text{ in } \text{max}(b, \text{findFirst}(a, b))
\end{aligned}$$

$$\begin{aligned}
\text{findFirst} &: x \rightarrow \overline{tuple} \rightarrow expr \\
\text{findFirst}[a, ts] &= \text{if } ts = \emptyset \text{ then } 0 \text{ else let } t \in ts \text{ in } t_{attributes}[a]
\end{aligned}$$

Figure 3.11: Translation rules for the aggregation functions

transitive closure for other data domains but the existence of possible holes would complicate the generation of the right equality constraints. Because of this reason we decided to postpone the calculation of transitive closure for other domains to possible future work.

In essence, calculating the transitive closure can be performed by recursively joining the relation with itself. To calculate the transitive closure we apply a variant of *iterative squaring* that works on our relation data structure. On every iteration the same translations are applied to the previously calculated relation. A single step of this translation is applied in the declared but not defined *joinOnce* function.

3.4.3 Testing the Translation

To gain more confidence in the correctness of the ALLEALLE translation we compared ALLEALLE with KODKOD [Tor09] and the CHOCO solver [PFL17] on a number of different Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP). KODKOD is also a relational model finder but is not able to solve optimization problems directly. The CHOCO solver is able to solve optimization problems but its formalism and solving strategy is semantically further away from ALLEALLE.

In the comparison of CSP problems we compare whether ALLEALLE and KODKOD, or ALLEALLE and CHOCO find the same answer, and whether they produce the same number of satisfying instances. When comparing COP problems we check whether both ALLEALLE and CHOCO find the same optimal solution. All problems

Table 3.1: Testing ALLEALLE against KODKOD and CHOCO.

Problem	Problem type	Compare with	Sat. ALLEALLE?	Sat. other?	#inst ALLEALLE	#inst other	Same opt. solution?
FileSystem	CSP	KODKOD	Yes	Yes	2184	2184	-
Handshake	CSP	KODKOD	Yes	Yes	24	24	-
Pigeonhole	CSP	KODKOD	No	No	-	-	-
RingElection	CSP	KODKOD	Yes	Yes	2	2	-
RiverCrossing	CSP	KODKOD	Yes	Yes	2	2	-
8Queens	CSP	CHOCO	Yes	Yes	92	92	-
Sudoku	CSP	CHOCO	Yes	Yes	1	1	-
SendMoreMoney	CSP	CHOCO	Yes	Yes	1	1	-
Knapsack	COP	CHOCO	Yes	Yes	-	-	Yes
Mariokart	COP	CHOCO	Yes	Yes	-	-	Yes

are existing examples or benchmark problems from KODKOD or CHOCO^{††}. The results are shown in Table 3.1.^{‡‡}

As can be seen in the results ALLEALLE finds the same solutions as KODKOD and CHOCO for all implemented problems. Please note that the reported found instances also contain all symmetric solutions. For instance the 8 queens problem has 92 solutions, but if symmetry is taken into account only 12 distinct solutions remain. KODKOD can detect such symmetries by generating so called *symmetry breaking predicates* but for this benchmark we configured KODKOD not to do this [Tor09].

3.5 EVALUATION

We evaluate ALLEALLE in terms of performance and expressiveness, by comparing the translation and solving times of ALLEALLE with KODKOD [Tor09] on different problems, and by implementing a real-world use case, optimal dependency resolution [ATC⁺11], respectively. Finally, we qualitatively compare ALLEALLE to similar systems for solving constraint problems.

3.5.1 Translation and Solving Time Benchmark

We compare ALLEALLE’s translation and solving time performance against the translation and solving time performance of KODKOD by translating and solving six different problems. Table 3.2 characterizes the benchmark problems in terms of the kinds of constraints that are used.

For five of these problems we measure the performance for different configurations of the same problem to get insight in the effect of the size of a problem specification. Configuration parameters are the number of atoms or tuples that are allowed to

^{††}See <https://github.com/chocoteam/samples/>

^{‡‡}See <https://github.com/joukestoel/allealle-benchmark/> for the encoding of the problems.

Table 3.2: Overview of the benchmarked problems.

Problem	Constraint types
Alloy FileSystem	Relational
Halmos handshake	Relational, cardinality
Pigeonhole	Relational
River crossing puzzle	Relational
Square $y = x^2$	Integer
Account state transition system	Relational, integer

populate the relations, and the allowed bit-width for the integer encoding used for `KODKOD`.

The benchmarks are run on a early 2015 MacBook Pro with a 2,7 GHz quad core Intel i5 processor with 8GB of DDR3 RAM. Java 8 (version 1.8.0_131 by Oracle) is used for all benchmarks. The translation and solving per configuration per problem was run 30 times with a warmup of 10 runs. All caches were flushed between each run. We report the median of the translation and solving times.

`ALLEALLE` is implemented in Rascal [KvdSV09]. Rascal is a functional programming language designed for the development and analysis of programming languages. It is an interpreted language that runs on the JVM. All `ALLEALLE` benchmarks were run using version 0.12.0.201901101505 of Rascal and version 4.8.0 of `Z3`.

`KODKOD` is implemented in Java and was built and run using the previously mentioned Java version. Some of the benchmarked problems contain cardinality and integer constraints. To avoid wrap-around semantics for integer constraints in `KODKOD` we use `KODKOD*` [MJ14; MNK⁺15], which is currently packaged with `ALLOY 4.2`; the solver is configured with `SAT4J` (version 2.3.5.v20130525). As earlier, we configured `KODKOD` to not generate symmetry breaking predicates.

Interpreting the results Table 3.3 contains the results of benchmarking `ALLEALLE` against `KODKOD`, comparing translation times and solving times. As can be seen in the results, `ALLEALLE` is slower in translating purely relational problems (e.g., `FileSystem`, `Pigeonhole` and `Rivercrossing`). The reason for the slow-down is twofold. First, our current implementation of `ALLEALLE` is a prototype, built as a proof-of-concept to demonstrate the correctness of the translation algorithm. Furthermore, `ALLEALLE` is implemented in Rascal, which is an interpreted language for language prototyping, whereas `KODKOD` is implemented in Java. We are currently working on a Java implementation of `ALLEALLE` which, we expect, will bring the translation performance up to par with `KODKOD` since `ALLEALLE` translation algorithms are of the same complexity as `KODKOD`'s translation.

`ALLEALLE` is also slower in solving purely relational problems compared to `KODKOD`. This can be explained from the fact that `ALLEALLE` generates more

clauses than KODKOD. For instance, for the FileSystem problem in the configuration with 30 atoms ALLEALLE generates a total of 5 359 296.00 clauses while KODKOD merely generates 24 753.00 clauses. The reason for this difference is that KODKOD implements a clause rewriting system that is much more aggressive than what is currently implemented in ALLEALLE [Tor09].

The results show, however, that ALLEALLE's native handling of data for problems that contain both relational and integer constraints, pays off, both in translation times and solving times. As mentioned before, KODKOD needs to specifically encode the possible integers up to the configured bit-width. This is needed because it needs to encode integer constraints as part of the SAT formula so that the underlying SAT-solver can solve the problem. This results in more clauses in the generated SAT formula and thus higher translation and solving times. Since ALLEALLE does not require this explicit encoding but can use the solvers built-in reasoning power on different theories it does not suffer from the same performance penalty. Therefore its translation and solving times is consistent for the same problem even if larger integer values are required.

For problems which encode explicit cardinality constraints like the HandShake problem we also see better performance of ALLEALLE since relations with higher cardinality require a larger bit-width in KODKOD.^{§§}

^{§§}Explicit meaning using KODKOD's integer cast expression `sum()` on relations and using integer arithmetic expressions to formulate cardinality constraints.

Table 3-3: Benchmark comparison between ALLEALLE and KODKOD. Comparison shows six problems of which five are shown with different relation sizes, either in number of atoms or integer bit width used.

Problem	#Atoms	Bit width (KODKOD only)	Sat?	ALLEALLE				KODKOD			
				Trans. time (in ms)	Solve time (in ms)	#Vars	#Clauses	Trans. time (in ms)	Solve time (in ms)	#Vars	#Clauses
FileSystem ^r	15.00	-	SAT	725.00	20.00	130.00	21839.00	12.00	4.00	135.00	3235.00
FileSystem ^r	30.00	-	SAT	14567.00	110.00	460.00	5359296.00	29.00	8.00	470.00	24753.00
HandShake ^{r,c}	10.00	4	SAT	480.00	10.00	184.00	1633.00	12.00	61.00	200.00	9292.00
HandShake ^{r,c}	17.00	5	UNSAT	2235.00	69865.00	548.00	6295.00	23.00	136465.00	578.00	39214.00
Pigeonhole ^r	9.00	-	UNSAT	51.00	10.00	20.00	302.00	2.00	1.00	20.00	137.00
Pigeonhole ^r	17.00	-	UNSAT	77.00	1180.00	72.00	1424.00	1.00	45.00	72.00	565.00
River crossing ^r	12.00	-	SAT	327.00	10.00	74.00	1621.00	7.00	0.00	68.00	749.00
Square ⁱ	2.00	4	SAT	5.00	10.00	2.00	6.00	2.00	4.00	36.00	3025.00
Square ⁱ	2.00	10	SAT	5.00	10.00	2.00	6.00	198.00	11883.00	2052.00	371722.00
Account ^{r,i}	12.00	5	SAT	72.00	10.00	33.00	289.00	21.00	79.00	351.00	19698.00
Account ^{r,i}	12.00	9	SAT	71.00	10.00	33.00	289.00	460.00	34080.00	5151.00	480198.00

Problem contains ^r relational constraints, ^c cardinality constraints, ⁱ integer constraints

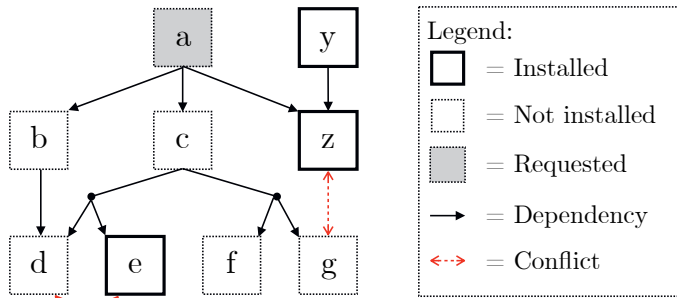


Figure 3.12: A package resolution problem (solution: install $\{a, b, c, d, f\}$ and uninstall $\{e\}$)

3.5.2 Optimal Dependency Resolution

To evaluate the expressiveness of ALLEALLE we have implemented a solution to the optimal package resolution problem, which is common in package managers like NPM, APT, or MAVEN [ATC⁺11]. Figure 3.12 shows an example of a package resolution problem taken from Tucker et.al. [TSJ⁺07]. The user asks to install package a ; the package resolver needs to compute which packages to install or uninstall in such a way that all dependencies are satisfied and no conflicts are violated. This has been shown to be an NP-complete problem [Cos05].

We ran the experiment on the “paranoid” track of the MISC 2012 competition; a solver competition for package managers.^{¶¶} This track required the contestants to find the minimal ‘change’ needed to the system to comply to the package update request.^{***} Because this problem requires to reason about the minimal change to the system it can not be directly encoded in ALLOY or KODKOD since these formalisms do not allow for the encoding of optimization problems.

All benchmarks were run on the same early 2015 MacBook Pro with a 2,7 GHz quad core Intel i5 processor with 8GB of DDR3 RAM. We compare our found results against the reported solutions and solving times from the MISC 2012 competition.

The package resolution problem can be compactly defined as a relational problem with data. The relevant relations are summarized in Table 3.4. The first 9 relations represent known facts about the package repository, what is installed on the user’s system, and what the user requests to install, remove, or upgrade, respectively. These relations have exact bounds meaning that for every possible solution the tuples of these relations are exactly those that are defined in the specification. The other 4 relations have an upper bound and represent the solution space for the solver to

^{¶¶}<http://www.mancoosi.org/misc-2012/>

^{***}Minimal change meaning, minimal amount of packages that need to be removed and the minimal amount of packages needed to be installed or updated.

Table 3.4: Relations for the optimal package resolution problem

Relation	Signature	Bound
installRequest	pId: id , relop: int , version: int	Exact
removeRequest	pId: id , relop: int , version: int	Exact
upgradeRequest	pId: id , relop: int , version: int	Exact
version	vId: id , pId: id , nr: int	Exact
installed	vId: id	Exact
keep	kId: id , vId: id	Exact
depends	vId: id , dcId: id	Exact
dependChoice	dcId: id , pId: id , version: int , relop: int	Exact
conflicts	vId: id , pId: id , version: int , relop: int	Exact
toBeInstalled	vId: id	Upper
toBeRemovedVersion	vId: id	Upper
toBeChanged	vId: id	Upper
toBeRemovedPackage	pId: id	Upper

satisfy the user’s request, given dependency and conflict constraints between package versions. The rest of the specification is shown in Listing 3.2, consisting of a mere 30 source lines of ALLEALLE code.

In total we translated 57 of the competition problems, and measured the time spent in translating ALLEALLE and running Z3 to solve the constraints. All found solutions by ALLEALLE were correct and optimal, showing that the constructed specification is a correct implementation of the optimal dependency resolution problem according to the paranoid criteria. The full results can be found in Table E.1, in Appendix E.1.

The results show that Z3 can efficiently solve the formula produced by ALLEALLE, in the same order of magnitude as the winning solving times from the 2012 MISC competition. On the other hand, the time spent by ALLEALLE translating the specification to SMT formula is high, ranging from 47 seconds to 62 minutes. Based on the specific problems exhibiting this behavior, we hypothesize that translation time correlates with the number of dependencies between the packages. The more dependencies between packages, the longer it takes ALLEALLE to translate the problem to SMT formulas.

3.5.3 Comparing ALLEALLE to Similar Systems

ALLEALLE is a constraint solving system and language, comparable to KODKOD, ALLOY, and SMT solvers. ALLEALLE is an *intermediate* language: it is higher-level than the first-order logic formulas of SMT solvers like Z3 [DB08] or CVC4 [BCD⁺11] which are more general purpose logic solvers, but lower-level than, e.g., ALLOY, which is an end-user, modeling language.

KODKOD is the back-end framework of ALLOY, which is at the same level of abstraction as ALLEALLE. Both ALLEALLE and KODKOD support relational

```

1 // All packages that are requested to be installed or upgraded should be part of the installation afterwards
2 let installedAfter = (toBeInstalled + (installed - toBeRemovedVersion)) |
3 (forall ir : installRequest | some (ir |x| version |x| installedAfter)
4   where ((relop = 0) || (relop = 1 && version = nr) || (relop = 2 && version != nr)
5     || (relop = 3 && nr >= version) || (relop = 4 && nr <= version)))
6 &&
7 (forall ur : upgradeRequest | some (ur |x| version |x| installedAfter)
8   where ((relop = 0) || (relop = 1 && version = nr) || (relop = 2 && version != nr)
9     || (relop = 3 && nr >= version) || (relop = 4 && nr <= version)))
10
11 forall rr : removeRequest | some (rr |x| version |x| toBeRemovedVersion)
12 // all the removal requests should be scheduled for removal
13 where ((relop = 0) || (relop = 1 && version = nr) || (relop = 2 && version != nr) || (relop = 3 && nr >= version) ||
14   (relop = 4 && nr <= version))
15
16 let installedAfter = (toBeInstalled + (installed - toBeRemovedVersion)) |
17 // installing version means installing its dependencies afterwards
18 forall d : depends | (d[vId] in installedAfter) =>
19   let possibleInstalls = ((d |x| dependChoice)[pId,version,relop] |x| (version |x| installedAfter)) |
20     (some (possibleInstalls where ((relop = 0) || (relop = 1 && nr = version) || (relop = 2 && nr != version) ||
21       (relop = 3 && nr >= version) || (relop = 4 && nr <= version))[vId] & installedAfter))
22
23 let installedAfter = (toBeInstalled + (installed - toBeRemovedVersion)) |
24 // when a version is installed, no conflicting version can be installed
25 forall c : conflicts | (c[vId] in installedAfter) =>
26   let possibleConflicts = (c[pId,version,relop] |x| (version |x| installedAfter)) |
27     no (possibleConflicts where ((relop = 0) || (relop = 1 && nr = version) || (relop = 2 && nr != version) ||
28       (relop = 3 && nr >= version) || (relop = 4 && nr <= version))[vId] & installedAfter))
29
30 let installedAfter = (toBeInstalled + (installed - toBeRemovedVersion)) |
31 // all versions to be kept need to be installed afterwards as well
32 forall k : keep | some k |x| installedAfter
33
34 toBeRemovedPackage = (toBeRemovedVersion |x| version)[pId] - (toBeInstalled |x| version)[pId]
35 toBeChanged = (toBeInstalled + toBeRemovedVersion)
36
37 objectives: minimize toBeRemovedPackage[count()], minimize toBeChanged[count()] // the paranoid criteria

```

Listing 3.2: Optimal Package Resolution in ALLEALLE

constraints, yet ALLEALLE employs Codd’s relational algebra making it possible to constraint data attributes directly (using the `where` operator), whereas KODKOD is based on Tarski’s relation logic allowing constraints only to be expressed on the level of relations.

Although in terms of abstraction level, ALLEALLE is comparable to KODKOD, the latter only exists as a Java library and does not feature a concrete syntax. ALLEALLE’s syntax allows us to experiment with specifications in a more flexible way, and thus may function as code generation target for higher-level languages, – essentially fulfilling the same role that KODKOD fulfills for ALLOY.

ALLEALLE leverages built-in theories of underlying SMT solvers, including support for optimization criteria available in solvers like Z3 [BPF15]. KODKOD (and hence ALLOY) require bit-encoding of integers because of their underlying SAT solvers, which is an impediment to performance for constraint problems that require such constraints. Optimization criteria are not available in either ALLOY or KODKOD. For instance, the optimal dependency resolution problem (Section 3.5.2) cannot be expressed in either ALLOY or KODKOD.

ALLEALLE thus occupies a sweet spot in terms of both expressiveness (Codd’s algebra) and solving performance (because of native SMT theories) between high-level languages like ALLOY, and low-level relational solvers like KODKOD.

3.6 RELATED WORK

SEM- and MACE-style model finders There are several finite model finding tools for first order logic (FOL). They can roughly be divided into two different groups: tools that implement specialized search strategies to find satisfying models (also known as SEM-style model finders) [Sla94; ZZ95] and tools that translate FOL formulas to SAT or SMT formulas and use an off-the-shelf solver to find satisfying instances (also known as MACE-style model finder) [CS03; McC94; Sag15; TJ07a]. ALLEALLE falls in the MACE-style category.

Although all of these model finders accept FOL formulas as input not many of them accept relational logic. SEM [ZZ95] and FINDER [Sla94] for instance accept a many-sorted logic of uninterpreted functions but no quantifiers. MACE2 [McC94], PARADOX [CS03], Fortress [VD16] and Razor [Sag15] do allow for quantifiers but do not offer direct support for relational expressions.

Razor opts for a slightly different angle where it focuses on model exploration by searching for a minimal model first and allowing for model exploration using a predefined preorder relation. To achieve this Razor also exploits the SMT solver Z3 [Sag15].

Fortress [VD16] also exploits an SMT solver by mapping FOL formulas to the logic of equality with uninterpreted functions (EUF). To make the model finite Vakili et al. introduce so-called *range functions* to force restriction on the number of elements assigned to the different sorts [VD16].

KODKOD [TJ07a], the model finding engine used by ALLOY [Jac12; TD06], accepts relational logic and transitive closure but offers no support for optimization criteria or first-class reasoning over data. As shown in the evaluation (see Section 3.5) KODKOD and ALLOY do support integers but require the user to specify a fixed bit-width. KODKOD uses an explicit integer encoding which results in the introduction of more variables and ultimately into more CNF clauses in the generated SAT formula.

The other main difference with KODKOD is the interpretation of relational logic: KODKOD uses Tarski’s definition of relational logic, ALLEALLE uses Codd’s relational algebra. The switch from Tarski to Codd prevents the need for an explicit encoding of data variables (i.e., integers) in the problem domain. e.g., in ALLEALLE it is not needed to explicitly define the set of integers to be used in the relational expressions, it can directly encode constraints on the data attributes (using the **where** operator).

Relational reasoning with SMT El Ghazi et al. translate ALLOY specifications to unbounded SMT constraints using an axiomatization of ALLOY’s relational logic in

SMT, a so called shallow embedding of a relational theory [ET11]. Although this allows for proving some ALLOY specifications it also struggles with many specifications, as is shown in later work [MRT⁺17].

Meng et al. define a relational calculus based on the theory of finite sets with cardinality constraints [BRB⁺16; MRT⁺17]. This calculus is created explicitly to be implemented as a new theory in SMT solvers, a so called deep embedding, and contains many of the relational expressions that are also part of KODKOD (like join, transpose, product and transitive closure). The calculus does not require that bounds are set on the relations but uses earlier work by the authors on finite model finding [RTG⁺13]. The authors implement the calculus as a new theory in the SMT solver CVC4 [BCD⁺11] and evaluate its performance on existing benchmarks. Contrary to ALLEALLE, the new theory is able to prove when a relational problem is unsatisfiable. Like ALLOY and KODKOD, ALLEALLE requires bounds on the relations with the result that any reported *unsat* only means that it is unsatisfiable with respect to the given bounds. The work by Meng et al. is based on the same calculus as KODKOD (Tarski's relational logic) meaning that reasoning on values of other theories (like integers) is limited in the same way as it is limited in KODKOD. Next to that, this work is implemented in CVC4 which currently does not support optimization objectives.

3.7 CONCLUSION

Relational model finding is a powerful technique that can be applied to a wide range of problems. In this paper we have introduced ALLEALLE, a new language for describing such problems with first-class support for data attributes. ALLEALLE specifications combine first-order logic with Codd's relational algebra, transitive closure, and optimization objectives. We have presented the formal semantics of ALLEALLE and a detailed exposition of a novel algorithm to translate ALLEALLE specifications to SMT constraints, to be solved by standard SMT solvers such as Z3.

Initial evaluation of our prototype implementation has shown that the performance leaves room for improvement. Both the translation speed and the generated formula efficiency can be improved. Theoretically it should be possible to improve the translation and solving time of pure relational problems to the level of KODKOD's performance since the translation algorithm of both model finders is comparable in terms of complexity. Early performance experiments with a pure Java version of ALLEALLE indeed hint into this direction.

ALLEALLE supports compact encodings of relational problems with data. One example is optimal dependency resolution, as used in package managers. We have used this problem to assess expressiveness of ALLEALLE. Although currently the translation times are high for these problems, the resulting SMT formulas can be solved efficiently by off-the-shelf solvers. The specification itself is concise, taking

only half a page, which shows the expressiveness of combining relational logic, data, and optimization criteria.

There are multiple directions for future work. The current prototype of ALLEALLE only supports integer domains; we will extend ALLEALLE with support for reals, strings, and bit vectors since most SMT solvers have built-in theories for these types. Another direction for further work is to include symmetry breaking predicates [CS03] to the generated SMT formulas, since this is well-known to have a positive effect on solver performance. Torlak et al. introduce a method to create these symmetry breaking predicates for relational logic [TJ07a], but it is an open question how this should be done in the presence of ALLEALLE's data attributes. A final direction for further research is to investigate how *unsatisfiability core extraction* [TCJ08] can be used to explain reasons for UNSAT results. A particular challenge is how to map such explanations back to the level of ALLEALLE.

CONSTRAINT-BASED RUN-TIME STATE MIGRATION FOR LIVE MODELING

Abstract

Live modeling enables modelers to incrementally update models as they are running and get immediate feedback about the impact of their changes. Changes introduced in a model may trigger inconsistencies between the model and its run-time state (e.g., deleting the current state in a statemachine); effectively requiring to migrate the run-time state to comply with the updated model. In this paper, we introduce an approach that enables to automatically migrate such run-time state based on declarative constraints defined by the language designer. We illustrate the approach using `NEXTEP`, a meta-modeling language for defining invariants and migration constraints on run-time state models. When a model changes, `NEXTEP` employs model finding techniques, backed by a solver, to automatically infer a new run-time model that satisfies the declared constraints. We apply `NEXTEP` to define migration strategies for two DSLs, and report on its expressiveness and performance.

4.1 INTRODUCTION

Live modeling [vdSto13; vRvdS17] allows users of executable modeling languages to enjoy live, immediate feedback when editing their models, without having to restart execution. Live modeling has its roots in live programming, which allows developers to change program code and use the running application simultaneously, without long edit-compile-restart cycles [McD13; RRL⁺19]. When the programmer changes the program code, its execution state is updated accordingly on-the-fly. This effectively bridges the “gulf of evaluation” [LF95] between changing a program and the impact of these changes.

While the value of live modeling and live programming is widely recognized [KRB18; LF95; Tan13], engineering live software languages requires a lot of effort and a deep understanding of the host language and its particular domain of application [BFdH⁺13].

A central problem for live modeling languages is how to reconcile changes to a model with the run-time state of its execution. Changes to a model or program might invalidate the current run-time state. That is, when the executing model is modified, its run-time state still corresponds to the version before the change. The problem is analogous to migrating a database after a change to the database schema. So the question we address in this paper is how to migrate run-time state after a change to an executing model.

Earlier work has explored (re)constructing the required migration steps to the run-time state from the changes applied to the model [vRvdS17]. Unfortunately,

this requires intimate operational knowledge of how the change of the model itself is propagated to the runtime in the first place, leading to brittle, imperative, and non-modular code.

In this work, we employ model finding techniques [MGC13] to migrate the runtime state of an executing model. A declarative, constraint-based specification of invariants and migration policies on the run-time state is input to a solver to infer a model that satisfies these constraints, given a representation of the old and new models, and the old run-time state. The resulting model is taken as the new run-time state of the updated version of the DSL program, and execution can continue.

We illustrate this approach using `NEXTEP`, a prototype meta-modeling language for defining the metamodel of both modeling language and run-time state model, and migration policies as constraints. `NEXTEP` specifications are then transformed to `ALLEALLE`, a language for relational model-finding modulo theories. The `ALLEALLE` specification is then processed by the `Z3` [DB08] constraint solver to obtain a new consistent run-time state instance.

The remainder of this paper is organized as follows. In Chapter 4.2, we introduce a motivating example of live modeling and detail the associated challenges. In Chapter 4.3, we present a framework for structuring the constraints related to run-time state migration. In Chapter 4.4, we introduce `NEXTEP`, our prototype language for live modeling. In Chapter 4.5, we evaluate the performance and expressiveness of `NEXTEP`. Finally, we discuss related work in Chapter 4.6 and draw some conclusions in Chapter 4.7.

4.2 MOTIVATING EXAMPLE

Unlike General-Purpose Languages (GPL), DSLs capture the syntax and semantics of a specific domain. As a result, the cognitive distance between DSL models and their dynamic behavior can be arbitrarily large. Live modeling can thus be considered especially beneficial for DSLs because the dynamic effect of a change in a DSL program is harder to predict.

Throughout this paper, we use a simple finite-state machine (FSM) DSL as a running example to explain the ideas and challenges of live modeling, and to illustrate our approach. A program in this DSL is a particular machine that consists of a number of states and transitions. An example scenario of live modeling for the FSM DSL is depicted in Chapter 4.1(a). The starting point (Step #0) is a simple machine with two states `closed` and `opened` and transitions between them. We choose to model the run-time state of machines as the combination of two elements: the current state of execution `current` and a map `visited` that keeps track of how many times each state has been visited. Note that the choice of what is part of the run-time state and how it is specified is up to the language designer. In particular, it can be specified as an extension of the static metamodel or as a separate semantic domain [CCP12]. Finally,

we assume the FSM DSL is executed by a step-based interpreter such that the user can trigger events to trigger transitions, and consequently, modify the run-time state. Changing the model pauses the interpreter for an instant so that the new version can be reloaded.

At Step #0, the current state is initialized to the initial state `closed` and every counter is initialized to 0. At Step #1, following the first step of execution, the user triggers an event, which changes the current state to `opened` and increments the `visited` counter for `closed` by one. At Step #2, the user decides to insert a new state `locked` in the currently running machine. As depicted in Chapter 4.1(a), the question that arises is: what is the new run-time state at this point? Does the newly added state `locked` become the current state? How should `locked` be captured in the `visited` map?

These questions get even more complicated when considering the live modeling scenario depicted in Chapter 4.1(b). In Step #2 of this scenario, the user chooses to delete the state `locked`, which is at this point the current state of the machine. Here an obvious question is: which state should be the new current state of the updated machine? Should it be reverted to the initial state of the machine?

A generic live modeling engine cannot provide answers to these questions, because they are inherently domain-specific and should thus comply with domain-specific run-time state migration policies. Ultimately, it is the language designer's responsibility to define these migration policies. We investigate how a language designer can express them in the next section.

Step	DSL program	Run-time state
#0: initial		current: closed visited: closed \mapsto 0 opened \mapsto 0
#1: interpreter step		current: opened visited: closed \mapsto 1 opened \mapsto 0
#2: user edit		current: ??? visited: ???

((a)) Live modeling scenario 'Add New State' for the FSM DSL

Step	DSL program	Run-time state
#0: initial		current: closed visited: closed \mapsto 0 opened \mapsto 0 locked \mapsto 0
#1: interpreter step		current: locked visited: closed \mapsto 1 opened \mapsto 0 locked \mapsto 0
#2: user edit		current: ??? visited: ???

((b)) Live modeling scenario 'Remove Current State' for the FSM DSL

When considering a situation of live modeling, as described in the previous section, we can distinguish two basic components: a *static model* (denoted p) and a *run-time model* (denoted x). The static model captures the structure (i.e., code) of the DSL program.* The run-time model captures the state of the execution of the DSL program. As we focus on the situation in which a DSL program is changed during its execution, we distinguish two versions of the static and run-time models: p and x before the program has been changed, and p' and x' after the program has been changed. Thus, we can reformulate the problem of finding a new run-time state as follows: given that p , x , and p' are known, how do we find x' ?

A potential approach would be to modify the model interpreter to repair the run-time state imperatively whenever the program is changed. Unfortunately, as we discuss in more detail in Section 4.5.3, this approach is error-prone and non-modular. In this paper, we apply model-finding techniques to infer the run-time state based on a set of declarative constraints. Concretely this means identifying requirements for x' and representing these as a set of constraints in relation to p , x , and p' , and then solving the constraints for x' . We introduce the following two categories (subsets) of constraints.

Semantic relations $R(p', x')$: these constraints ensure that the new run-time state x' is semantically consistent with the current (new) version of the DSL model p' . Note though that there might be multiple—possibly infinite— x' that are semantically consistent with p' .

Migration rules $M(p, x, p', x')$: these constraints provide the possibility to take into account what has changed in the DSL model and project this change onto the run-time state. Migration rules restrict the choice of (semantically consistent) x' in preference for those x' that reflect the change that has happened to the DSL (static) model.

The definition of a solution space by semantic relations $R(p', x')$ and its filtering by migration rules $M(p, x, p', x')$ is illustrated by the Venn diagram depicted in Figure 4.1. Although migration rules restrict the choice of possible solutions, there still can be many (potentially infinite) x' that satisfy the semantic constraints. Selecting an arbitrary element of this set as the new run-time state for the updated DSL program might not result in an intuitive new run-time state. For instance, in our previously described FSM scenario ‘Add New State’ (Table 4.1(a)), it is perfectly valid to reset the `current` to `opened`, and set the `visited` entities to 1000. Although this would be a correct run-time state (i.e., a valid solution to our set of constraints), it would only add confusion to the user of the DSL.

*Although according to the main idea of live modeling the static model is not static anymore, we still use the term *static* for the sake of the terminology legacy.

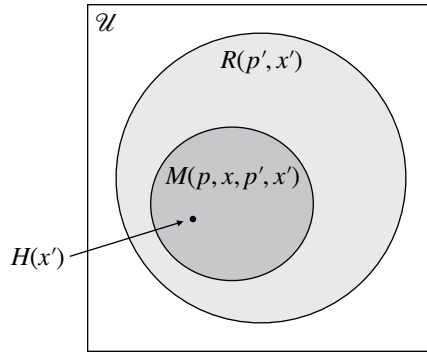


Figure 4.1: Partitioning and selection of the new run-time model (x')

To find the most suitable new run-time state, we need to introduce a heuristic for selecting one of potentially many solutions. We represent such a heuristic as $H(x')$ in Figure 4.1. What are good heuristics may depend on many variables related to domain-specific semantics and usability. One example of a heuristic that we will use in the remainder of this paper is the *minimum distance* $D(x, x')$. The minimum distance heuristic requires that the new run-time state x' should be as close as possible to the previous run-time state x . Intuitively, this heuristic captures the principle of “first, do no harm” (i.e., do not break parts of the run-time state that do not have to change), and the “principle of least surprise” (i.e., prefer small incremental changes over invasive ones). In this way, we support the incremental process of editing a DSL model, providing live feedback in relation to the previous run-time state of the DSL program.

To apply constraint-based run-time state migration, language designers must specify the semantic relations $R(p', x')$ and migration rules $M(p, x, p', x')$. Semantic relations can be automatically extracted from the DSL definition itself (metamodel, static and dynamic semantics). Migration constraints are explicitly specified to relate p, p', x , on the one hand, and x' , on the other hand.

In the case of the FSM DSL, if the user chooses to delete the state pointed to by `current` (Table 4.1(b)), a possible strategy is to reset `current` to the initial state of the machine. Another strategy is to set the `current` to the state that was the “closest” to the deleted state. Both policies are equally valid: this is the designer’s responsibility to make a choice. In contrast, the heuristic $H(x')$ is domain-agnostic and can be built into the live modeling tool itself.

Note that our approach infers the new run-time state (x') only based on the directly preceding step of the execution (x), i.e., migration constraints cannot be defined in terms of the full execution trace. For example, in our FSM DSL we cannot assign the

new current to a state that had been current previously in the execution. Taking the execution trace into account is an essential direction for further research.

4.4 NEXTEP: A LANGUAGE FOR STATE MIGRATION

In this section we describe our proof-of-concept implementation of the approach described in Section 4.3, the NEXTEP language. In particular, we discuss all described constraints in detail and explain their implementation in NEXTEP for our running example of the FSM DSL.

4.4.1 Syntax of NEXTEP

The NEXTEP language allows for formulating preferences for (i.e., for configuring) live modeling in the form of a set of constraints. As described earlier, such constraints are expressed using the constructs of the static and run-time models of a DSL.[†] For this, NEXTEP uses a simplified version of the Ecore metamodeling language to express classes and references between them. An example NEXTEP definition for the statemachine DSL is depicted in Listing 4.2.

A NEXTEP configuration consists of three parts: **static**, **runtime**, and **migration**. The **static** part (lines 1–10 in Listing 4.2) describes the static model of the DSL and, in essence, corresponds to (the core of) the DSL metamodel. In our example, it consists of three classes (`Machine`, `State`, and `Trans`) and four references (`states`, `initial`, `transitions`, and `target`).

References can be of two types: referencing exactly one object (for example, `target: state`, line 10 in Listing 4.2) or referencing a set of objects (such as `states: State*`, line 3 in Listing 4.2). However, this is not a conceptual limitation of NEXTEP, but rather a simplification for the sake of demonstration. For the sake of conciseness, we also omit static constraints in the statemachine metamodel from the NEXTEP specification, and assume that these are dealt with by other means, such as parsers or type checkers.

The **runtime** part of a NEXTEP definition (lines 12–27 in Listing 4.2) describes the run-time model of the DSL and the semantic relations $R(p', x')$. The run-time model is defined in the same way as the static model, using classes and references. The semantic relations are defined in the form of invariants attached to these classes.

The class `Runtime` is a root class that stores references to all components that constitute both the static and the run-time models of the DSL. For the FSM DSL, these are `machine` for the static model, and `current` and `visited` for the run-time model (lines 14–16 in Listing 4.2). This way, the `Runtime` class defines the scope for the semantic relations $R(p', x')$. We define the following semantic relations for the FSM DSL:

- the current state of the machine is one of its states (line 19 in Listing 4.2);

[†]See Appendix B.3 for the definition of the concrete syntax of NEXTEP.

```

1  static
2    class Machine
3      states: State*
4      initial: State
5
6    class State
7      transitions: Trans*
8
9    class Trans
10     target: State
11
12  runtime
13    class Runtime
14     machine: Machine
15     current: State
16     visited: Visit*
17
18     invariants
19     current in machine.states
20     forall s: machine.states | one (visited.state & s)
21     forall v1:visited, v2:visited | v1 != v2 => v1.state != v2.state
22
23    class Visit
24     state: State
25     nr: int
26
27     invariant: nr >= 0
28
29  migration
30  not (old.current in new.machine.states) => new.current = new.machine.initial

```

Figure 4.2: NEXTEP specification of the FSM DSL.

- the number of visits is defined exactly once for each state of the statemachine (line 20 in Listing 4.2);
- different entries in `visited` correspond to different states of the statemachine, `visited` is a function (line 21 in Listing 4.2);
- the number of visits for each state is greater than or equal to zero (line 27 in Listing 4.2).

The `migration` part of a NEXTEP definition (lines 29–30 in Listing 4.2) describes the migration rules $M(p, x, p', x')$. Here the keywords `old` and `new` denote the instances of the `Runtime` class corresponding to the old and new versions of the run-time model, respectively. Since the `Runtime` class refers to the static model, this also provides

access to p and p' . For the FSM DSL, we define the following migration rule: if the current state has been deleted (i.e., `not (old.current in new.machine.states)`), then the new current state is assigned to the initial state (i.e., `new.current = new.machine.initial`).

4.4.2 Semantics of NEXTEP

In order to implement model finding for our live DSLs, we translate a NEXTEP definition of a DSL and its static and run-time models into ALLEALLE. ALLEALLE[‡] is a *bounded relational model finder*. That is, ALLEALLE is a formalism that allows for expressing a model using a combination of first order logic and relational algebra and supports automatic construction of relational instances of such a model in a bounded scope (i.e., limited set of cases) based on the constraints of the model. In addition, ALLEALLE supports declaration of optimization criteria making it suitable for solving optimization problems next to pure satisfiability problems.

As a back-end ALLEALLE uses Z₃, an SMT (Satisfiability Modulo Theories) solver [DB08].[§] That is, ALLEALLE translates a model expressed in its input language into a corresponding SMT formula and then invokes Z₃ to find solution(s) to this formula. The solution is then translated back into an instance of the relational model.

ALLEALLE is comparable to the Alloy language and analyzer [Jac12].[¶] The major difference is that, unlike Alloy which is built on top of a SAT solver, ALLEALLE is built on top of the Z₃ SMT solver, and thus, supports unbounded integer numbers and allows for optimization criteria, which are used by the solver to find the most suitable (optimal) solution.

In this section, we describe the semantics of NEXTEP in the form of a translation to ALLEALLE. We illustrate our description using an excerpt of the ALLEALLE specification (see Listing 4.1) that corresponds to the NEXTEP specification described in the previous section (Listing 4.2).

Structure translation. All structural elements of both static and dynamic parts of a NEXTEP definition are translated to relations in ALLEALLE:

- NEXTEP classes are translated to ALLEALLE unary relations;
- NEXTEP references are translated to ALLEALLE binary relations.

As the resulting ALLEALLE specification is used for solving our initial problem “given p , x , and p' , what is $x'?$ ”, all the listed relations are generated for the concrete instances of p , x , p' , and x' .

For example, in Listing 4.1 relations with `pn_` as their prefix (lines 1–5) correspond to the new version of the program p' ; relations with `xo_` as their prefix (lines 7–8)

[‡]<http://github.com/cwi-swat/allealle>

[§]<http://github.com/Z3Prover/z3>

[¶]<http://alloytools.org>

```

1 pn_Machine (mId: id)           = {<doors>}
2 pn_State (sId: id)            = {<closed>, <opened>, <locked>}
3 pn_states (mId: id, sId: id)  = {<doors, closed>, <doors, opened>,
4                                <doors, locked>}
5 pn_initial (mId: id, sId: id) = {<doors, closed>}
6 ...
7 xo_current (mId: id, sId: id) = {<doors, opened>}
8 xo_visited (sId: id, val: int) = {<closed, 1>, <opened, 0>}
9 xn_current (mId: id, sId: id) <= {<doors, closed>, <doors, opened>,
10                                <doors, locked>, <doors, s1>..

```

Listing 4.1: An excerpt of the generated ALLEALLE code for the statemachine DSL

correspond to the previous run-time state x ; and relations with $xn_$ as their prefix (lines 9–13) correspond to the new run-time state x' .

Bounds translation. As mentioned earlier, bounds are used to restrict the scope of search for the back-end solver. In particular, bounds introduce a set of atoms available for the instantiation of relations in an ALLEALLE specification. In the context of NEXTEP, p , x , and p' are known and, thus, determine the exact bounds for the corresponding ALLEALLE relations. In other words, each of the relations that represent the original p , x , and p' are assigned specific values (i.e., tuples) corresponding to the concrete instances of p , x , and p' (lines 1–8 in Listing 4.1, on the right side of the '=' symbol).

While p , x , and p' are known, the new run-time model x' is what we are searching for. Therefore, bounds for the ALLEALLE relations representing x' define an extended search scope, not limited to the concrete instances of p , x , and p' . In particular, the x' relations are constrained by an upper bound (lines 9–13 in Listing 4.1, on the right

side of the ' \leq ' symbol). The upper bounds include the atoms from the instances of p , x , and p' , and a finite number of auxiliary atoms of the same type. For instance, in the example, this means that there is one extra machine `m1` and five extra states `s1..s5`.

Constraints translation. Semantic relations and migration rules of a `NEXTEP` definition are translated to the corresponding `ALLEALLE` constraints (formulas) instantiated for p , x , p' , and x' (lines 17–24 in Listing 4.1). Moreover, additional `ALLEALLE` constraints are generated to capture the structural properties of (static and runtime) classes: types and multiplicities of references (line 15 in Listing 4.1). The expressive power of `NEXTEP` constraints is determined by the expressive power of `ALLEALLE` formulas, which includes relational algebra extended with transitive closure. Thus, `NEXTEP` supports set theory notation, first-order predicates, (reflexive) transitive closure, and unbounded integer arithmetic. For object navigation we use the standard dot notation.

Minimum distance. As described earlier in Section 4.3, the heuristic for the minimum distance $D(x, x')$ is not specific to each particular DSL. In particular, in `ALLEALLE`, the minimum distance heuristic is generically represented as optimization criteria. For this, $D(x, x')$ is represented as a set of metrics in `ALLEALLE`, over which the minimization criteria are applied (lines 23–24 in Listing 4.1).

Minimization is performed over the following metrics:

- the number of elements in the difference of two sets representing x and x' correspondingly (such as `current` in the `statemachine` DSL);
- the sum of absolute differences between integer values of two vectors representing x and x' correspondingly (such as `visited` in the `statemachine` DSL).

As a result, for instance, not modifying the `current` state is preferred over modifying it, because in the former case the set difference will be the empty set, which is minimal. Similarly, minimization of integer values ensures that no arbitrary values will be put in the map `visited`.

4.4.3 Output of `NEXTEP`

When applied to the two live modeling scenarios described in Section 4.2, `NEXTEP` produces the following results. Tables 4.2 and 4.3 show the instances of p , x , and p' and the corresponding x' for the scenarios of Table 4.1(a) and Table 4.1(b), respectively.

Here, the instances of x' (right bottom corner in both tables) represent the solutions found by the solver. They are constructed by mapping an instance of the relational model (calculated by `Z3` and translated to `ALLEALLE`) on the corresponding instance of the `NEXTEP` run-time model.

Table 4.2: NEXTEP migration for the statemachine scenario ‘Add New State’

p states: closed, opened transitions: (closed \mapsto t1), (opened \mapsto t2) target: (t1 \mapsto opened), (t2 \mapsto closed)	x current: opened visited: closed \mapsto 1 opened \mapsto 0
p' states: closed, opened, locked transitions: (closed \mapsto t1, t3), (opened \mapsto t2), (locked \mapsto t4) target: (t1 \mapsto opened), (t2 \mapsto closed), (t3 \mapsto locked), (t4 \mapsto closed)	x' current: opened visited: closed \mapsto 1 opened \mapsto 0 locked \mapsto 0

Table 4.3: NEXTEP migration for the statemachine scenario ‘Remove Current State’

p states: closed, opened, locked transitions: (closed \mapsto t1, t3), (opened \mapsto t2), (locked \mapsto t4) target: (t1 \mapsto opened), (t2 \mapsto closed), (t3 \mapsto locked), (t4 \mapsto closed)	x current: locked visited: closed \mapsto 1 opened \mapsto 0 locked \mapsto 0
p' states: closed, opened transitions: (closed \mapsto t1), (opened \mapsto t2) target: (t1 \mapsto opened), (t2 \mapsto closed)	x' current: closed visited: closed \mapsto 1 opened \mapsto 0

As can be seen in Table 4.2, the addition of the `locked` state does not change the runtime state, except for adding a new entry to the `visited` map, initialized to zero. However, if the current state is removed (i.e., `locked` in Table 4.3), the `current` field is reset to the initial state, and the `locked` entry is removed from the `visited` map.

The obtained results are determined by the domain-specific migration policies in the NEXTEP definition (in Listing 4.2). For example, if we remove the migration rule from our NEXTEP definition, the calculated x' for the scenario ‘Remove Current State’ would be some arbitrary state (e.g., `opened`), instead of the initial state of the statemachine.

4.5 EVALUATION

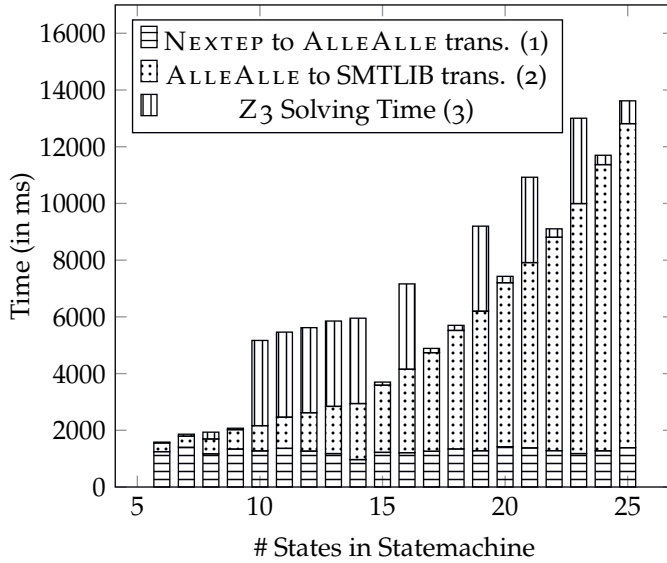


Figure 4.3: Scenario 'Add New State'. Z3 timed out for 10,11,12,13,14,16,19,21 and 23

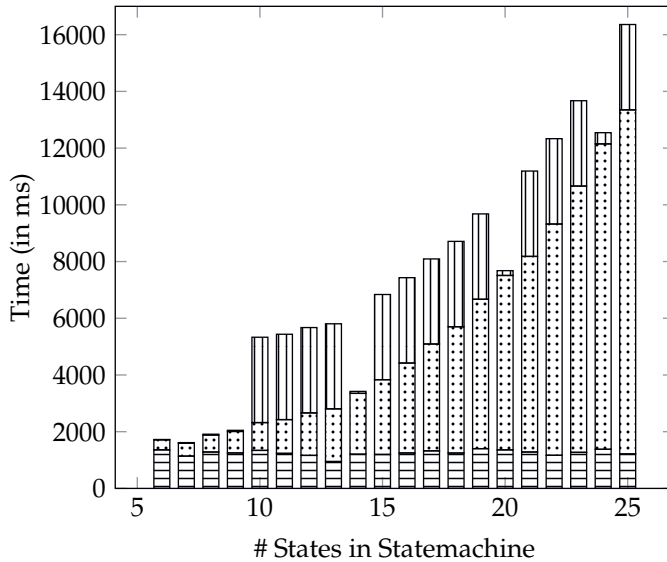


Figure 4.4: Scenario 'Remove Current State'. Z3 timed out for 10,11,12,13,15,16,17,18,19,21,22,23 and 25.

In this section, we use the NEXTEP implementation for an early evaluation of our approach. In Section 4.5.1, we perform a quantitative evaluation to estimate the performance of NEXTEP. In Section 4.5.2, we perform a qualitative evaluation to challenge the expressive power of NEXTEP by applying it to a DSL for robotic arm control. Finally, in Section 4.5.3, we conceptually compare NEXTEP with the manual encoding of migration strategies as presented in previous work [vRvdS17].

4.5.1 *Statemachine DSL Case Study*

To evaluate the performance of our method, we ran two different scenarios for the FSM example.

In the first scenario, we tested the live modeling situation where one extra state and two new transitions are added to the new program. The corresponding new run-time state has its `visited` map extended with an entry for the newly added state assigned to 'o' visits. In the second scenario, we tested the case where the `current` state is removed from the statemachine. This forces the new `current` to be reset to the `initial` keeping all the values for the `visited` map intact, minus the removed state.

In both scenarios, we gradually increased the number of states in the statemachine to assess the impact of model size on performance. The results are shown in Figure 4.3 and Figure 4.4. The execution times reported in the figure include the following three phases of our implementation: (1) translation of the NEXTEP specification to the relational specification, (2) translation of the relational specification to SMT constraints (from ALLEALLE to SMTLIB), and (3) solving (model finding) by Z3.

We ran the benchmarks on an early 2015 model MacBook Pro with a 2.7 GHz quad core Intel processor with 8 GB of DDR3 RAM. We ran each configuration of the benchmark 10 times, ensuring that each execution is independent of the other (i.e., no shared memory, cache, etc.). Figure 4.3 and Figure 4.4 report the mean time per configuration of each of the three different phases.

The translation times reported in both scenarios show a similar pattern. The translation of NEXTEP specifications to ALLEALLE specifications in the different configurations is fairly stable time-wise with a minimum of 943 ms and a maximum of 1529 ms.

The translation times from ALLEALLE to SMTLIB also follow a similar trend in both scenarios: the translation time goes up as the NEXTEP specification grows. An explanation is that increasing the size of NEXTEP specifications causes the number of tuples in the ALLEALLE relational models to increase as well. Then, the more tuples in an ALLEALLE specification, the longer it takes to translate from ALLEALLE to SMTLIB. This is especially the case when quantifiers are used in the ALLEALLE specifications—and NEXTEP relies heavily on these quantifiers.

The source code of our benchmark experiments can be found at <https://github.com/cwi-swat/live-modeling/tree/master/nextstep/src/benchmark/statemachine>

The solving times reported for Z_3 tell a different story. For many configurations, Z_3 was not able to find an optimal solution within a time limit of 3 seconds. We empirically set this limit to 3 seconds after observing that Z_3 either manages to find a solution within a second or quickly explodes to multiple minutes. Since the time frame of multiple minutes is not in line with the spirit of live modeling, we decided to consider all solving times exceeding 3 seconds as timeouts.

For the first scenario, Z_3 timed out for 9 configurations out of 19. For the second scenario, it timed out for 13 configurations out of 19.

Discussion Our benchmark results suggest that there is much room for improvement in our prototype. These improvements can be classified according to the different phases of our method: improving the translation from `NEXTEP` to `ALLEALLE`, improving the translation from `ALLEALLE` to `SMTLIB`, and improving the solving times by Z_3 .

A first improvement would be to optimize the translation from `NEXTEP` to `ALLEALLE`. As mentioned earlier, the current implementation of `NEXTEP` is an early prototype that has not been aggressively optimized yet. Both the performance of the translation and the generated `ALLEALLE` specifications can be improved. In particular, improving the latter would help in speeding up the total execution time of our method. To verify this intuition, we optimized some `ALLEALLE` specifications manually, bypassing the automatic translation phase of our approach while keeping the exact semantics of the original `NEXTEP` specification, and observed substantial improvements on performance. Generating an optimized `ALLEALLE` specification would help in minimizing the translation time from `ALLEALLE` to `SMTLIB` as well, as we have observed that translation times decrease when the `ALLEALLE` specification shrinks. Finally, generating optimized `ALLEALLE` specifications would also help Z_3 to find optimal solutions quicker: the smaller the `ALLEALLE` specification the less variables are declared in the resulting SMT formula, often making it easier for the solver to find optimal solutions.

Next to optimizing `NEXTEP`, we could look at optimizing `ALLEALLE`. The current implementation of `ALLEALLE` is also in a prototype phase. Early experiments with an optimized native Java version of the `ALLEALLE` model finder suggest that translation times can potentially be improved by a factor 10 to 15.

Improving the stability of the solving times of Z_3 is the hardest challenge which for now is considered an open question. As mentioned earlier, generating optimized `ALLEALLE` specifications will help to improve the solving times for Z_3 . However, whether solving time can be predicted upfront from a given `NEXTEP` specification remains an open question.

4.5.2 Robotic Arm DSL Case Study

The *Robotic Arm DSL* is used for controlling industrial robots that consist of various subsystems (i.e., mechanical and/or electrical parts). For example, a robot can consist of a *hand*, responsible for manipulating objects, an *arm*, responsible for moving the hand to a particular position, and smaller subsystems like *triggers* and *actuators*. A program in the Robotic Arm DSL describes the coordination of subsystems of a robot during its execution. An example of such a program is shown in the middle column of Table 4.4.

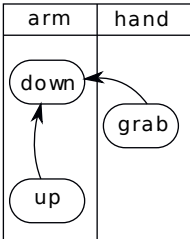
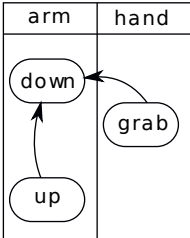
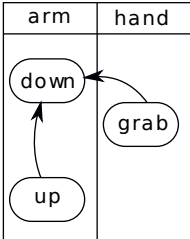
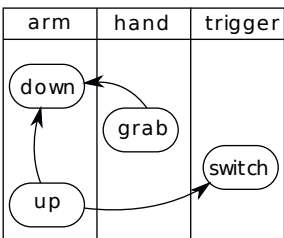
To perform a certain task, a robot executes specific actions on its subsystems. Each subsystem can execute its actions independently from other subsystems of the robot. However, some of these actions should be executed in certain order, or in other words, some actions have to wait for other actions to be executed first; such dependencies are shown as arrows in Robotic Arm programs. For example, the subsystem *hand* should not execute the action `grab` before the subsystem *arm* has executed the action `down` (see the DSL program in the second row of Table 4.4).

The semantics of the Robotic Arm DSL maps a Robotic Arm program to the following execution behavior. In order to execute actions independently, each subsystem maintains a queue of actions that should be executed. First, all actions are collected in one `Request` pool (Step 0 in Table 4.4). Then, the actions are taken from the `Request` one by one and pushed to the corresponding subsystem `Queue` (Steps 1–2 in Table 4.4). When an action is executed, it is removed from the queue. To enforce the ordering of actions, an action cannot be queued until all of the actions that this action depends on are in the `Request` pool or in the `Queues`. An exception to this constraint is the situation in which both ordered actions belong to the same subsystem. Then the actions can be queued in the corresponding order. For example, in Step 2 of Table 4.4, both actions `down` and `up` are queued, as they belong to the same subsystem *arm*.

The bottom row of Table 4.4 introduces a live modeling step for the Robotic Arm DSL. In particular, interrupting the normal execution of the Robotic Arm program (Steps 0–2), in Step 3 the user updates the program by adding a new subsystem *trigger* with a new action `switch` and a new dependency between the actions `up` and `switch`. As a result of this update, an obvious migration strategy for the run-time state would be to add a new queue for the subsystem *trigger*. It is not clear though what to do with the action `switch`. According to the semantics of the Robotic Arm DSL, the action `switch` can be executed independently by *trigger*, and thus, does not need to appear in the `Request` or `Queues`. However, intuition suggests that a newly added action should (explicitly) show up in the execution. Therefore, we introduce a migration rule that puts newly added actions to the `Request`.

The Robotic Arm `NEXTSTEP` definition that captures the described semantic relations and specifies the stated migration rule is depicted in Listing 4.5. The new run-time state calculated according to this `NEXTSTEP` definition is shown in the bottom right cell

Table 4.4: Update scenario for the Robotic Arm DSL

Step	DSL program	Run-time state
0		Request: {grab, down, up} Queues: arm \mapsto [] hand \mapsto []
1		Request: {grab, up} Queues: arm \mapsto [down] hand \mapsto []
2		Request: {grab} Queues: arm \mapsto [down, up] hand \mapsto []
3		Request: {grab, switch, up} Queues: arm \mapsto [down] hand \mapsto [] trigger \mapsto []

```

1  static
2  class Task
3    actions: Action*
4    order: OrderedPair*
5
6  class Subsystem
7    ssa: Action*
8
9  class Action
10
11 class OrderedPair
12   action: Action
13   succeeds: Action
14
15 runtime
16 class Queue
17   subsystem: Subsystem
18   actions: QueuedAction*
19
20 invariant:
21   forall qa: actions |
22     qa.item in subsystem.ssa
23
24 class QueuedAction
25   item: Action
26   index: int
27
28 class Runtime
29   request: Action*
30   queues: Queue*
31   task: Task
32
33 invariants
34   request in task.actions
35
36 forall s: Subsystem |
37   some (s.ssa & task.actions) =>
38     (exists q: queues | q.subsystem = s)
39
40 forall q: queues |
41   no (q.actions & request.actions)
42
43 forall o: task.order |
44   o.succeeds in request =>
45     o.action in request
46
47 forall o: task.order |
48   (exists q: Queue |
49     o.succeeds in q.actions.item
50     && not (o.action in q.subsystem.ssa) ) =>
51     not(exists q:Queue |
52       o.action in q.actions.item)
53
54 forall o: task.order |
55   (exists q: Queue |
56     o.succeeds in q.actions.item &&
57     o.action in q.actions.item) =>
58     (forall qa1: queues.actions,
59       qa2: queues.actions |
60       qa1.item = o.succeeds &&
61       qa2.item = o.action
62       => qa1.index > qa2.index)
63
64 migration
65   (new.task.actions -- old.task.actions)
66   in new.request.actions

```

Figure 4.5: NEXTEP definition for the Robotic Arm DSL

of Table 4.4. Note that in this new run-time state, the newly added action `switch` is added to the `Request` and, as a result, the action `up` is moved (back) to the `Request`. This allows for observing the execution order according to the newly added dependency.

The NEXTEP definition of Listing 4.5 defines classes for all constructs introduced above: `Task` for a Robotic Arm program, `Subsystem` for a robot subsystem, `Action` for a subsystem action, `OrderedPair` to represent order dependencies, `Queue` and `QueuedAction` to model a subsystem queue, and `Runtime` for the run-time state.

The semantics of the Robotic Arm DSL is captured by the following semantic relations:

1. a `Queue` contains only actions of its corresponding subsystem (line 21–22);

2. a `request` contains only actions of the current task (i.e., program being executed) (line 33);
3. if an action of a subsystem appears in the `task`, then there should be a corresponding queue for this subsystem (lines 35–37);
4. `queues` and `request` do not intersect (line 39–40), i.e., when an action is queued, it is removed from the `request`;
5. an action, that succeeds another action which is stored in the `request`, should be also in the `request`, i.e., should not be queued yet (lines 42–44);
6. an action, that succeeds an action stored in a `queue`, should not be queued yet, unless these actions belong to the same subsystem (lines 46–51);
7. if two ordered actions belong to the same subsystem queue, then their `indexes` correspond to the order of the actions (lines 53–61).

Finally, the result that we get for the scenario of Table 4.4 is to a large extent determined by the migration rule specified in lines 64–65 of Listing 4.5 according to our earlier design decision. In particular, the constraint states that all actions from the new task (`new.task.actions`) that are not in the old task (`-- old.task.actions`) should be added to the request (`in new.request.actions`).

To conclude, in this section we demonstrated how `NEXTEP` can be applied for configuring live modeling for a DSL with a more complicated semantics than our running example of the FSM DSL. The obtained result (i.e., the calculated new run-time state) corresponds to our expectations about the semantic consistency of the DSL and to our intuition about migration strategy of the run-time state.

4.5.3 Comparison to Manual Migration

Earlier work in live modeling explored an operational way of encoding migration strategies, called `RMPATCH` [vRvdS17]. `RMPATCH` consists of the following steps:

1. after a user changes a DSL program (static model), a delta is computed using a model-based differencing algorithm;
2. a patch corresponding to the delta is then applied to the run-time model, which is a copy of the static model extended with the necessary run-time data;
3. the code that applies the patch to the run-time model is specialized using inheritance and updates the run-time data.

One of the take-aways of the experiment with `RMPATCH` is that even for simple examples such as the statemachine language, it is quite hard to correctly implement migrations, and the code quickly becomes unwieldy. Below we consider the main drawbacks of manually coded migrations in contrast with `NEXTEP`.

Coupling between model and run-time state. `RMPATCH` requires that run-time state is an instance of the runtime meta-model which is a proper extension of the

static meta-model of the language. For instance, the `State` class defines the number of visits as its own field. Similarly, the class `Machine` contains the reference to the current state. Using `NEXTEP` the representation of a run-time state is decoupled from the meta-model, thus allowing for more flexibility for languages where the relation between model and run-time state is less clear cut.

Operational instead of declarative. Second, using program code to encode migration policies requires careful scheduling of operations, since there might be dependencies between modification incurred by the user's model change and migration side effects. In contrast, `NEXTEP`'s migration rules are defined as declarative constraints, thus allowing for abstracting away from any ordering constraints.

Hard to reason about correctness. The third problem of encoding migration policies manually using code is that migration is interleaved with the update of (the copy of) the static model within the run-time model. As a result, the run-time model is in an inconsistent state itself when the migrations are being applied. Separating migration logic until after the update has fully finished is non-trivial, since migrations depend on the knowledge of what has changed (i.e., the delta itself).

Scattering of migration logic. Another problem is that migration logic often requires expressing and maintaining global invariants on the run-time state. Modularizing the code according to the type of model elements (e.g., `State`, `Transition`, etc.), or delta operations (e.g., `Add`, `Remove`, etc.), causes scattering of migration logic over multiple classes and methods.

Lack of extensibility. Finally, as a result of the previous point, migration logic is not modularly extensible. Extending the DSL with new constructs and corresponding additional invariants and rules requires invasive modification of the existing migration code. With `NEXTEP`, migration rules and invariants can simply be added to the `NEXTEP` definition, i.e., in conjunction with existing constraints. The potential interactions between constraints are handled by the solver back-end.

To summarize, `NEXTEP` improves upon the earlier migration work in that it provides a declarative, decoupled, and modular way of expressing migration policies. The heavy lifting of finding the new run-time state that is consistent with the new version of the DSL program is delegated to the solver.

4.6 RELATED WORK & DISCUSSION

In this section, we discuss the related work from three different points of view: the problem of run-time state migration (Section 4.6.1), the solutions based on constraint

solving (Section 4.6.2), and the area of DSL debugging (Section 4.6.3). Next to highlighting the existing approaches, we discuss which of their findings we can adopt in our work in order to improve efficiency, expressiveness, and usability of our approach.

4.6.1 *Model Synchronization*

In this paper, we address the problem of finding a new run-time state that is consistent with an updated DSL program. In a broader perspective, this is a particular case of the problem of keeping different models in sync, which includes well-known sub-problems: consistency checking between different models, change propagation, co-evolution, model repair, etc. As such, this problem has been studied in different research fields and from multiple angles. For instance, the field of views and viewpoints engineering studies the problem of consistency checking between source models and views. The corresponding approaches that address this problem include incremental backward change propagation [SDH⁺16], lenses [FGM⁺07], and triple-graph grammars [Sch94]. Our approach distinguishes itself in letting the language designer specify her own migration policies in a declarative way.

In Model-Driven Engineering (MDE), consistency relations between different models are typically defined through model-to-model transformations. Such model transformations can be used to (re)construct a new (run-time) model for an updated (static) model. Concretely, our work was inspired by the work of Macedo et al. [MC13; MGC13], which implements QVT-R bidirectional model transformations using Alloy and its SAT solver to construct consistent models. In comparison, in our work we use ALLEALLE and the SMT solver as a back-end, which allows for more expressiveness when configuring live modeling for a DSL.

Bill et al. use UML and OCL to formalize a synchronization model that includes both a change model and a consistency model [BGW16]. The synchronization model defines inter-model relations and constraints (comparable to our NEXTEP definition); the change model introduces all possible changes that can be used to construct a new model and assigns a cost to each of such changes. The USE model finder uses these cost values to find the most suitable (optimal) model. On the one hand, defining all possible changes (including language-agnostic, language-specific, and composite changes) requires operational thinking and can result in a tangled and complex model. In our approach, we strive towards a declarative definition. On the other hand, using cost values the authors define five different heuristics for finding the most suitable solution (including the least change strategy that we employ in NEXTEP). Thus, we plan to leverage their experience in future work to extend NEXTEP with additional heuristics.

4.6.2 *Constraint-based Solving*

In our approach, we use the SMT solver to find a new run-time model that satisfies the specified constraints. Recently, constraint solvers are being adopted in MDE for the automatic analysis of modeling languages. For instance, Erata et al. present AlloyInEcore, a meta-modeling language that allows for adding Alloy-like invariants into Ecore metamodels [EGK⁺18]. Built on top of the Kodkod (SAT) solver, their tool automatically detects inconsistent models and completes partial models. As both AlloyInEcore and NEXTEP translate to the relational logic (of Kodkod and ALLEALLE correspondingly), the syntaxes of these two languages are similar. As AlloyInEcore is a more developed and mature language comparing to NEXTEP, we consider learning from the design decisions of AlloyInEcore in our future work.

Straeten et al. assess the performance of using constraint solvers (Kodkod in particular) for resolving inconsistencies in models [SPM11]. They experiment with models that were reverse-engineered from a set of open-source projects and further translated to Kodkod, to which they add consistency rules formulated directly in Kodkod. The model sizes range from 2064 to 8572 elements. The results obtained in this work demonstrate that “the approach does not provide instantaneous resolution on medium scale models”. These results suggest a potential threshold for our approach. However, we believe that we can still significantly improve the performance of NEXTEP.

An alternative approach to employing SAT/SMT solvers is to use constraint programming systems. For instance, Cabot et al. translate UML class diagrams extended with OCL constraints into Constraint Satisfaction Problems (CSPs) and the ECLiPSe constraint programming system to construct model instances [CCR14]. Steimann et al. use constraints embedded into an attribute grammar to check the well-formedness of programs and to compute repair alternatives for malformed programs [SHU16]. Both these works propose various optimizations and search algorithms in order to improve performance of constraint-based model finding.

4.6.3 *DSL Debugging*

Although the objectives are different, live modeling is also closely related to debugging. Debuggers enable programmers to monitor the execution of a program, set breakpoints, inspect and modify runtime values, and, under certain constraints, hot swap some parts of the code itself. Over time, a number of debuggers have been developed for modeling languages, for instance for DEVS [vMvTV17], fUML [MLK12], or individual diagrams of the Unified Modeling Language (e.g., [CD08; DK07]).

Beyond classical forward-in-time debuggers, Bousse et al. proposed a methodology for the development of generic omniscient debuggers for DSLs [BLC⁺18] backed by efficient and domain-specific execution trace management facilities [BMC⁺17]. Ráth

et al. use the VIATRA [VB07] framework to simulate Petri nets. Users can edit models on-the-fly at simulation time, for instance to resolve cases of non-determinism [RVV08].

We expect many potential cross-fertilization between model debugging and live modeling, and we would like to investigate in the future how our constraint-based approach can solve some of the current problems of debuggers, regarding, for instance, migration of run-time state after code swapping.

4.7 CONCLUSION & FUTURE WORK

Conclusion Live modeling has the potential to improve the experience of using executable DSLs. Immediate feedback allows DSL users to better assess the impact of the changes they make to their models. A central problem for live modeling languages is how to reconcile changes to a model with the run-time state of its execution. In this paper, we proposed to formulate the problem of run-time state migration in terms of constraints on the run-time state.

We have identified two categories of such constraints: *semantic relations* which ensure that the new run-time state is consistent with the new version of the DSL model, and domain-specific *migration rules* which are specified explicitly by the language designer. To choose the most suitable run-time state out of potentially many solutions, we used the heuristic of *minimum distance* between the new and old run-time states.

We have illustrated this approach using NEXTEP, a meta-modeling language that allows to define such invariants and migration policies. NEXTEP employs model finding technique, backed by a solver, to automatically infer a new run-time model that satisfies the declared constraints.

We have evaluated the performance of NEXTEP on a simple FSM DSL. Initial results show that performance is satisfactory, albeit unpredictable even for similar problem specifications. Furthermore, we have applied NEXTEP to an existing DSL for robotic arm control, which is semantically richer than the FSM DSL. Overall, our results show that constraint-based state migration as realized by NEXTEP is a promising technique for engineering live modeling languages.

Future Work A number of research questions stem from our early prototype and evaluation which we hope to address in future work.

First, we did not address all the steps of the live modeling experience. Once a new run-time state is inferred, how should it be provided back in a seamless way to the user? How should it be provided to the DSL interpreter itself to let it resume the execution transparently?

Second, we believe user experience plays a crucial aspect in live modeling. How should a user interact with the live modeling framework, through which interface? To be adopted, live modeling must be as close to real time as possible to provide a

seamless experience. How to optimize the time it takes to find a new run-time state? We believe that live modeling tools should be evaluated empirically with real users to assess their benefits, and we hope to address these questions in the future.

Finally, we consider the use of a generic minimum distance heuristic to guide the constraint solving process as both a strength and a weakness of our approach. Investigating whether other generic heuristics or domain-specific heuristics could be employed requires more experience with various kinds of DSLs and remains future work.

Abstract

Writing formal specifications often requires users to abstract from the original problem. Especially when verification techniques such as model checking are used. Without applying abstractions the search space the model checker need to traverse tends to grow quickly beyond the scope of what can be checked within reasonable time.

The downside of this need to omit details is that it increases the distance to the implementation. Ideally, the created specifications could be used to generate software from (either manual or automatic). But having an incomplete description of the desired system is not enough for this purpose.

In this work we introduce the REBEL2 specification language. REBEL2 lets the user write full system specifications in the notation of state machines with data without the need to apply abstraction while still preserving the ability to verify non-trivial properties. This is done by allowing the user to *forget* and *mock* specifications only when running the model checker. The original specifications are untouched by these techniques.

We compare the expressiveness of REBEL2 and the effectiveness of *mock* and *forget* by implementing two case studies: one from the automotive domain and one from the banking domain. We find that REBEL2 is expressive enough to implement both case studies in a concise manner. Next to that, when performing checks in isolation, mocking can speed up model checking significantly.

5.1 INTRODUCTION

Formal specifications combined with model checking have been shown to be very effective in capturing and verifying desired system behavior. However when applying model checking, the user is forced to think about the potential state space the model checker needs to traverse [CHV⁺18]. Not taking this into enough consideration will lead to a state space that is too large to check within reasonable time.

Applying *abstractions* is one of the most used techniques to overcome the state space explosion problem [CGK⁺99; GM14]. Coming up with an abstraction that models the problem in sufficient detail but is abstract enough such that it can be model checked is in most cases left up to the specifier. Because of this need of omitting details the specifications become incomplete. Translating such incomplete specification to code, either automatic or manual, becomes very difficult since the software that ultimately needs to run does need to contain all the omitted details.

The methods that do allow to specify systems until the level of executable code such as Event-B [AH07] often require proofs that show that a refinement of an abstract

specification is indeed true to its abstraction. Writing such specifications and proofs can increase the required effort both in time and expertise [Den09]. This effort can be considered as too high and might be one of the reasons that industry adoption of these techniques, aside from safety critical systems, is as of yet still low.

In this work we approach this problem from another angle, by combining formal specifications, model checking and *mocking*. Mocking is a well known testing technique from object-oriented programming where mock objects are created to replace domain code with a dummy implementation for the purpose of emulating its behavior [MFCoo]. These mock objects are passed to the objects under test to allow for the testing of features in isolation with respect to the rest of the system. Similarly we propose to use *mock specifications* to replace specified behavior with a dummy specification for the purpose of model checking. This allows to perform model checks for parts of the system, isolated from other specifications. Although this technique is inherently unsound we expect that it can still be of value. It allows the user to make a pragmatic tradeoff between completeness of the check versus timely feedback from the checker. This type of trade-off is not new. For instance, the lightweight formal method Alloy is based around the "small-scope" hypothesis which says "many bugs are found in a small scope" favoring partiality over completeness as well, although in a different dimension [Jaco2b].

We implement this mocking technique in REBEL2, a lightweight formal specification method.* REBEL2 specifications use the notation of state machines with data and guarded transitions. Assumptions and assertions can be expressed using Linear Temporal Logic (LTL). To check assertions REBEL2 uses a bounded model checking technique. This is realized by translating the specifications to the relational algebra of ALLEALLE [SvdSV19] which in turn translates it to SMT and uses the Z3 SMT solver [DB08].

When model checking the user can mock parts of the specifications by using two different language constructs: **forget** and **mock**. The **forget** construct slices out data and constraints on this data. The **mock** construct replaces a specification entirely with a mock specification. Like mock objects, mock specification can emulate parts of the original specification in isolation which in turn can potentially reduce the state space considerably. Although we introduce the constructs **forget** and **mock** in the context of the REBEL2 specification language, the ideas are more general and could be implemented in other state-based techniques as well.

To test the expressiveness of REBEL2 and the effectiveness of model checking with mocking we evaluate REBEL2 on two different case studies, one from the automotive domain and one from the financial domain.

To summarize, the contributions of our work are:

*The term *lightweight formal method* was coined by Daniel Jackson in 2001 [JW96]. It describes methods with emphasis on partiality (partiality in language, modeling, analysis and composition).

1. A description of the lightweight formal specification language REBEL2 by example (Section 5.2).
2. A formalization of the **forget** and **mock** constructs (Section 5.3).
3. A prototype implementation of REBEL2 and model checking with mocking using **forget** and **mock** (Section 5.4).
4. An evaluation of our technique both in expressiveness and effectiveness (Section 5.5).

We conclude this chapter with a discussion of related work (Section 5.6) and possible future work (Section 5.7)

5.2 REBEL2 BY EXAMPLE: MONEY TRANSFER

In this section we introduce REBEL2 by specifying a simple bank account state machine.[†] REBEL2 is inspired by earlier work of Stoel et al. [SSV⁺16].[‡]

Accounts can be opened, and after an initial deposit, any number of deposit, withdraw and pay interest events are possible. An account can be temporarily blocked (e.g., in case of suspicious transactions) and unblocked. When an account is blocked, no transactions are allowed. Eventually an account can be closed either normally or by force.

Listing 5.1 shows the REBEL2 specification of an account that complies with the rules stated above. A REBEL2 specification consists of four parts, *fields*, *events*, *states*, and *assumptions*. We explain the first three below. Assumptions are discussed further-on in the section.

Fields Fields represent the internal state of a state machine. The Account specification declares three fields. Fields can have primitive types, e.g., *balance* (of type **Integer**). But fields may also refer to other REBEL2 specifications as a type, as is the case with the *nr* field, which has type *AccountNumber*.

Events Events define the business events and actions that may be triggered on a state machine. In the Account example there are eight. Events may have formal parameters (e.g., *amount* in *deposit* and *withdraw*), and are (optionally) guarded by preconditions. The effect of an event is specified in the form of a postcondition where the next value of a field is accessible by priming its name. For instance, the effect of *withdraw* is defined as **this**.*balance*' = **this**.*balance* – *amount*, effectively decrementing the account's balance.

[†]See Appendix B.4 for the full syntax of REBEL2.

[‡]Section 6.4.1 in Chapter 6 describes the differences between REBEL and REBEL2 in more detail.

```

1 spec Account
2   nr: AccountNumber, balance: Integer, openedOn: Date;
3
4   init event open(nr: AccountNumber, openedOn: Date)
5     post: this.balance' = 0, this.nr' = nr,
6         this.openedOn' = openedOn;
7
8   event deposit(amount: Integer)
9     pre: amount > 0;
10    post: this.balance' = this.balance + amount;
11
12   event withdraw(amount: Integer)
13     pre: amount > 0, this.balance >= amount;
14    post: this.balance' = this.balance - amount;
15
16   event payInterest(rate: Integer)
17     post: this.balance' = this.balance + ((this.balance * rate) / 100);
18
19   final event close()
20     pre: this.balance = 0;
21
22   event block()
23   event unblock()
24   final event forceClose()
25
26   states:
27     (*)      -> activation: open;
28     activation -> opened: deposit;
29     opened    -> opened: deposit, withdraw, payInterest;
30     opened    -> blocked: block;
31     blocked   -> opened: unblock;
32     blocked   -> (*): forceClose;
33     opened    -> (*): close;

```

Listing 5.1: REBEL2 specification of an Account.

States The last section in Listing 5.1 defines the lifecycle of an account, by defining state transitions of the form “*from* -> *to*: *via*,...”, where *via* is a list of events declared earlier. The special marker (*) is used to indicate initial and final states. Events from the initial state need to be marked as initial (cf. init **event** open), and events to the final state have to be marked **final** (cf. close and forceClose).

Checking assertions Now that we have specified our Account we can validate its behavior by checking the safety property that an account can not be overdrawn. For this, REBEL2 supports assertions. The above property is then expressed as follows:

```
assert CantOverdrawAccount = forall ac:Account |
    always (ac is initialized => ac.balance >= 0);
```

Assertions are expressed using Linear Temporal Logic (LTL) expressions. REBEL2 supports the standard LTL operators *always*, *eventually*, *next*, *until*, and *release*. The CantOverdrawAccount assertion can thus be read as follows: for all possible execution paths, all initialized accounts have a non-negative balance.

REBEL2 utilizes bounded model checking. To check the above property it is required to specify the bound in terms of i) the number of instances (e.g., Account objects) that take part and ii) the maximum search depth.

The number of instances is specified using the config directive:

```
config Basic = ac: Account is uninitialized,
              aNr: AccountNumber, dt: Date;
```

A configuration defines all the specification instances that can participate during model checking. This configuration specifies that there are three instances: an Account, an AccountNumber and a Date. All instances in a configuration are bound to a label (i.e., Account is bound to *ac*, AccountNumber to *aNr* and Date to *d*). The config statement supports constraining the state and field values of an instance. In the example the required state of the Account instance *ac* is set to uninitialized. The state of the AccountNumber and Date instances are not specified and thus are left open for the underlying model checker to decide.

The maximum search-depth is specified when invoking the verifier through the check command:

```
check CantOverdrawAccount from Basic in max 10 steps;
```

This instructs the model checker to try and find a counter-example to the property CantOverdrawAccount, starting in the Basic configuration, with a maximal search depth of 10 consecutively triggered event.

Forget Running the model checker on the above check command, results in a time-out, because the AccountNumber and Date specifications (not shown here) are complex state machines. As a result the state space that the checker must traverse is too large to find a counter example or exhaust within the default 30 second time-out.

The **forget** modifier can be used to abstract from the *nr* and *openedOn* fields in the Basic configuration, as follows:

```
config Basic = ac: Account forget nr, openedOn
    is uninitialized;
```

The result is that the field definitions and all constraints referencing these fields are removed resulting in a smaller (but well-formed) specification. Since the fields that reference the AccountNumber and **Date** specifications have been removed, the instances of these specifications can also be removed from the configuration (cf. aNr and dt).

Running the model checker again now results in a trace, a counter example for our assertion. The trace shows an execution path for which the assertion does not hold. In other words, it is possible to overdraw the account according to the specification. The following execution trace is shown: [§]

```
Counter example found:
1 (INIT): ac (Account) is 'uninitialized' :
  --> Raised open(nr = an) on ac (Account)
2: ac (Account) is 'activation' : balance = 0
  --> Raised deposit(amount = 1) on ac (Account)
3: ac (Account) is 'opened' : balance = 1
  --> Raised payInterest(rate = -101) on ac (Account)
4 (GOAL): ac (Account) is 'opened' : balance = -1
```

Examining the trace shows the root of the problem: the rate parameter of the payInterest action can be negative (see the third step in the trace). A way to prevent this is by adding the constraint `rate >= 0` to the precondition of the payInterest event:

```
event payInterest(rate: Integer)
  pre: rate >= 0;
  post: this.balance' =
    this.balance + ((this.balance * rate) / 100);
```

Rerunning the model checker after this fix yields the desired result: no counter example is found given the specified configuration and search depth.

Synchronization To illustrate synchronization between state machines, Listing 5.2 shows the specification of a Transaction which captures a transfer of money between two accounts. This is modeled in the book event, which triggers the withdraw and book events on the frm and to accounts, respectively. Semantically, all three events happen as a single atomic step: either all three succeed, or none.

To check whether it is possible to perform such a booking a *simulation* is run. The difference between a check and a simulation is that the model checker is not instructed to look for a counter example, but to find a witness of the assertion of interest instead.

Here's the assertion of interest:

```
assert CanBookATransaction =
  exists t: Transaction | eventually book on t;
```

[§]Traces can be shown both textual and visual. In this paper they are listed in their textual notation.


```

1 spec Transaction
2   frm: Account, to: Account, amount: Integer;
3
4   init event create(frm: Account, to: Account, amt: Integer)
5     pre: frm != to, amt > 0;
6     post: this.frm' = frm, this.to' = to,
7           this.amount' = amt;
8
9   final event book()
10    pre: this.frm.withdraw(this.amount),
11         this.to.deposit(this.amount);
12
13  states:
14    (*) -> created: create;
15    created -> (*): book;

```

Listing 5.2: Specification of a Money Transaction.

The assertion `CanBookATransaction` states that at some point in time there exists a Transaction on which the event `book` has been triggered.

Just like with `check`, a configuration specifies the elements participating:

```

config BasicTrans = t: Transaction is uninitialized,
  ac1,ac2: Account, an1,an2: AccountNumber,
  d1,d2: Date;

```

We use the `run` command to have the model checker find a witness:

```

run CanBookATransaction from BasicTrans in max 5 steps;

```

Executing this command causes a time-out because, like earlier, the state space is too large to check due to the inclusion of the detailed `AccountNumber` and `Date` specifications. Instead of slicing out fields from the participating instances (using `forget`), we want to keep the interaction between the Transaction and the two accounts in place since this is the essence of a transaction. To realize this, REBEL2 offers the `mock` operator to substitute simpler entities for certain instances in a configuration.

Mocking Similar to mock classes in object-oriented programming, mocking in REBEL2 entails writing a compatible specification that acts as a drop-in replacement for another specification. A potential mock specification of `Account` is shown in Listing 5.3.

This `MockAccount` contains two new concepts, internal events, and `assume`. The internal modifier signals to the model checker, that the event can not be triggered in isolation, but it can occur as part of a synchronizing event, like `book` in `Transaction`.

```

1 spec MockAccount
2   balance: Integer;
3
4   internal event withdraw(amount: Integer)
5     pre: amount > 0;
6     post: this.balance' = this.balance - amount;
7
8   internal event deposit(amount: Integer)
9     pre: amount > 0;
10    post: this.balance' = this.balance + amount;
11
12  assume PositiveBalance =
13    always forall a:MockAccount | a.balance >= 0;
14
15  states:
16    opened -> opened: withdraw, deposit;

```

Listing 5.3: A mock specification of the Account of Listing 5.1

Assumptions are invariants that the model checker assumes to always hold, expressed using the same LTL and FO formulas used in assertions. For instance, the PositiveBalance assumption in MockAccount allows the model checker to assume that balance is always non-negative, an assumption that we have checked earlier on actual accounts.

Mock specifications must be *compatible* with the mocked specification in that it needs to support the same events (including their signature) as the original, *restricted to* the events that are potentially triggered by the check.[¶] For instance, MockAccount is a valid mock specification for Account because it supports both events which can be triggered by the book event of Transaction, namely withdraw and deposit.

The mock specification can now be used in the definition of a configuration and run invocation:

```

config SimplifiedTrans =
  t: Transaction is uninitialized,
  ac1,ac2: MockAccount mocks Account;

run CanBookATransaction from SimplifiedTrans in max 5 steps;

```

Running the model checker returns the following witness showing a trace where a Transaction is booked:

[¶]Checking whether a mocked specification is compatible is in the current implementation of REBEL2 left to the specifier. Configuring a mocked specification that is not compatible will result in an error during translation.

Witness found:

```
1 (INIT): ac1 (Account) is 'opened' : balance = 7
  ac2 (Account) is 'opened' : balance = 1
  t (Transaction) is 'uninitialized' :
  --> Raised create(from = ac1, amount = 3, to = ac2) on t (Transaction)
2: ac1 (Account) is 'opened' : balance = 7
  ac2 (Account) is 'opened' : balance = 1
  t (Transaction) is 'created' : from = ac1, amount = 3, to = ac2
  --> Raised book() on t (Transaction) : affected instances {t,ac2,ac1}
3 (GOAL): ac1 (Account) is 'opened' : balance = 4
  ac2 (Account) is 'opened' : balance = 4
  t (Transaction) is 'finalized'
```

Unsoundness In this section we have introduced the formal modeling language REBEL2 and shown how the **forget** and **mock** constructs can be used to check and simulate properties of interest. Note, however, that both **forget** and **mock** are *unsound*: neither construct guarantees that the abstractions they create are equivalent with the original specification. This is by design. However, just like the “small scope” assumption used in tools such as Alloy, we conjecture that, nevertheless, it is possible and convenient to check non-trivial, useful properties, in limited amounts of (solving) time. This gives the user additional flexibility and freedom in defining checks and simulations, without immediately running into time-outs. As such, **forget** and **mock** introduce a pragmatic middle-ground between full formal verification and traditional testing as is practiced in many organizations.

5.3 FORMALIZATION

To define the semantics of **forget** and **mock** we need to define the semantics of REBEL2 as a framework. For this we will use the logic proposed in *State / Event Linear Temporal Logic* (SE-LTL) [CCO⁺04]. This logic contains both the notion of states and events and operates over a *Labeled Kripke Structure* (LKS). A LKS is a 7-tuple $(S, Init, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ where S is a finite set of states, $Init \subseteq S$ is the set of *initial states*, P is a finite set of *atomic propositions*, $\mathcal{L} : S \rightarrow 2^P$ is a *state labeling function* from states to atomic propositions, $T \subseteq S \times S$ is the transition relation, Σ a finite set (or *alphabet*) of *events* and $\mathcal{E} : T \rightarrow (2^\Sigma \setminus \{\emptyset\})$ the *transition labeling function*. We will write $s \xrightarrow{E} s'$ to denote a transition where $(s, s') \in T$ and $E \subseteq \mathcal{E}(s, s')$. If E is a singleton set we will just write $s \xrightarrow{e} s'$. The transition relation T is assumed to be *total* meaning that every state has a successor (no *deadlock* can occur).

A *path* $\pi = \langle s_1, e_1, s_2, e_2, s_3, \dots \rangle$ is an infinite alternating sequence of states and events in which for each $i \geq 1$, s_i and $s_{i+1} \in S$, $e_i \in \Sigma$ and $s_i \xrightarrow{e_i} s_{i+1} \in \mathcal{E}$. All paths together make up for the *language* of an LKS written as $L(M)$.

5.3.1 REBEL2 Specification to LKS

To map REBEL2 specifications to an LKS we use the following translation. The set of atomic propositions P contains all fields and possible values of a REBEL2 specification \mathcal{R} . For the sake of our formalization we will restrict the **Integer** and **String** domains to a bounded set of values where the bounds are arbitrary chosen. This way we restrict the set of states S and the set of atomic propositions P to be finite. The state as defined in a REBEL2 specification is also considered part of the set of atomic propositions and should not be confused with the state set S of the LKS. Also the parameters of the defined events in R are considered as being part of the set of atomic propositions.

The state labeling function \mathcal{L} maps values to fields for each possible $s \in S$. Σ contains all event labels as defined in R . We derive \mathcal{E} by calculating for each possible $s, s' \in S$ and event $e \in \Sigma$ the enabled events by checking whether the preconditions of e hold in $\mathcal{L}(s)$ and whether the postconditions hold in $\mathcal{L}(s')$.

5.3.2 Semantics of the Forget Operator

Our formalization of **forget** maps to the formalization of *abstraction* of an LKS as defined by Chaki et al. [CCO⁺04]. We will recall the notion of abstraction from [CCO⁺04] to show the meaning of **forget**.

Let $M = (S_M, Init_M, P_M, \mathcal{L}_M, T_M, \Sigma_M, \mathcal{E}_M)$ and $A = (S_A, Init_A, P_A, \mathcal{L}_A, T_A, \Sigma_A, \mathcal{E}_A)$. A is considered an abstraction of M (written $A \sqsubseteq M$) iff:

1. $P_A \subseteq P_M$.
2. $\Sigma_A = \Sigma_M$.
3. For every path $\pi = \langle s_1, e_1, s_2, e_2, \dots \rangle \in L(M)$ there exists a path $\pi' = \langle s'_1, e'_1, s'_2, \dots \rangle \in L(A)$ such that, for each $i \geq 1$, $e'_i = e_i$ and $\mathcal{L}_A(s'_i) = \mathcal{L}_M(s_i) \cap P_A$.

This is also known as *variable hiding* since an abstraction A contains a subset of the propositional variables of M while still accepting the original language of M .

This is also the essence of the **forget** operator. It will hide all atomic propositions bound to the field that it is instructed to forget (e.g., every references that maps a value to the field is removed from the set P). All constraints in the pre- and postconditions of the event referencing the forgotten field can be considered to evaluate to *true* and thus can be removed.

5.3.3 Semantics of the Mock Operator

To define the meaning of the **mock** operator we must first define the notion of *parallel composition* as defined by Chaki et al. [CCO⁺04]. We recall it here for clarity.

Parallel composition is defined via *shared events*. It is not allowed to share variables between two LKSs. This facilitates the possibility to perform compositional reasoning.

Let $M_1 = (S_1, Init_1, P_1, \mathcal{L}_1, T_1, \Sigma_1, \mathcal{E}_1)$ and $M_2 = (S_2, Init_2, P_2, \mathcal{L}_2, T_2, \Sigma_2, \mathcal{E}_2)$ then two LKSs are considered compatible if (1) they do not share any variables: $S_1 \cap S_2 = P_1 \cap P_2 = \emptyset$, and (2) their parallel composition yields a total relation (so that no deadlock can occur). Thus, the parallel composition can be defined as: $M_1 \parallel M_2 = (S_1 \times S_2, Init_1 \times Init_2, P_1 \cup P_2, \mathcal{L}_1 \cup \mathcal{L}_2, T, \Sigma_1 \cup \Sigma_2, \mathcal{E})$ where $(\mathcal{L}_1 \cup \mathcal{L}_2)(s_1, s_2) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$ and T and \mathcal{E} are such that $(s_1, s_2) \xrightarrow{E} (s'_1, s'_2)$ iff $E \neq \emptyset$ and one of the following holds:

1. $E \in \Sigma_1 \setminus \Sigma_2$ and $s_1 \xrightarrow{E} s'_1$ and $s_2 = s'_2$
2. $E \in \Sigma_2 \setminus \Sigma_1$ and $s_2 \xrightarrow{E} s'_2$ and $s_1 = s'_1$
3. $E \in \Sigma_1 \cap \Sigma_2$ and $s_1 \xrightarrow{E} s'_1$ and $s_2 \xrightarrow{E} s'_2$

To put it on other words, LKSs synchronize on shared events while proceeding independently from each other.

Communication between REBEL2 machines operates in a similar manner. The difference is that REBEL2 allows users to define which events must synchronize instead of relying on the mechanism of shared labels. Shared event parameters become part of the local variables of both machines to maintain the separation of variables and allow for compositional reasoning. State and field queries (between two specifications) can be interpreted as synchronized communication between two machines.

We also use this notion to define the **mock** operator. Assume we have two LKSs M_1 and M_2 having the same signature as described earlier. Also assume that M_1 and M_2 have shared events: $\Sigma_1 \cap \Sigma_2 \neq \emptyset$. A LKS $M_3 = (S_3, Init_3, P_3, \mathcal{L}_3, T, \Sigma_3, \mathcal{E})$ **mock** M_2 iff (1) the parallel composition of M_1 and M_3 is valid (no shared variables, composition yields a total relation) and (2) M_3 has the same shared events as M_1 and M_2 : $\Sigma_1 \cap \Sigma_3 = \Sigma_1 \cap \Sigma_2$. This means that all events that were synchronized in the original composition, $M_1 \parallel M_2$ will synchronize in the abstracted composition $M_1 \parallel M_3$.

5.3.4 On the Logic of SE-LTL

Using the earlier definitions of LKS we use the definition of State/Event Linear Temporal Logic as defined by Chaki et al. [CCO⁺04]. This logic operates on both states and events and has the following syntax: $\phi = p \mid e \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X} \phi \mid \mathbf{G} \phi \mid \mathbf{F} \phi \mid \phi \mathbf{U} \phi$ where $p \in P$ and $e \in \Sigma$. The operators **X** (next), **G** (always), **F** (eventually) and **U** (until) have their usual semantics,

Besides propositional constraints REBEL2 also allows for first-order constraints like quantification (forall, exists) and relational operators like membership (**in**). Given that

REBEL2 only operates in a bounded setting (i.e., a bounded number of machines and states) these operators can be translated to propositional logic via known translations (i.e., [Jac12; MBC⁺16; SvdSV19]).

5.4 IMPLEMENTATION

Performing a REBEL2 check (or run) follows a pipeline described in Figure 5.1. This pipeline consists of four steps: *prepare*, *normalize*, *translate* and *interpret*. The general scheme is that we translate REBEL2 specifications onto the relational algebra of ALLEALLE [SvdSV19] which in turn translates its relational algebra to an SMT formula and calls the Z3 SMT solver [DB08] to check whether the formula is satisfiable. If it is satisfiable the result is interpreted back into the domain of REBEL2, via ALLEALLE, where it is presented as an interactive visualization or textual trace to the user. REBEL2, the language and the transformations needed for model checking, are all implemented using the Rascal Language Workbench [KvdSV09]. We will now discuss each step in more detail.**

5.4.1 Step 1: Preparation

In the first step of the pipeline the assertion which is to be checked (or run) is prepared. As an example we will use the check that was formulated earlier which we recall here for readability:

```
check CantOverdrawAccount from Basic in max 10 steps;
```

This check references the CantOverdrawAccount assertion and the Basic configuration. The Account specification references the AccountNumber and **Date** specifications. To be able to perform checks on an account we must therefore at least also include the AccountNumber and **Date** specifications. Therefore the referenced Basic configuration is declared as:

```
config Basic = ac: Account is uninitialized,  
             aNr: AccountNumber, dt: Date;
```

For each check or run that is performed a specialized module is created during preparation. This module, using the syntax of REBEL2 (meaning that it is itself a valid REBEL2 module), contains exactly those specifications that are referenced in the config and specs reachable from the check or run command to be performed. This newly created module is *self contained* meaning that it does not need any external dependencies (i.e., imports) to be checked.

To be able to create a module that only contains those specifications that are referenced a specification dependency graph is build which in turn is used to find all

See <https://github.com/cwi-swat/rebel2>

**See Appendix D.1 for the complete translation of the earlier introduced Account example.

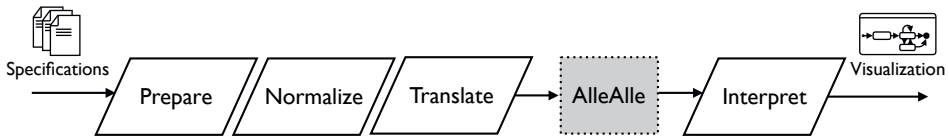


Figure 5.1: Overview of the model checking processing pipeline for REBEL2 specifications. The white parallelograms are part of REBEL2. The gray box is an external process.

reachable specifications. The reachability algorithm checks which specifications are reachable from those specifications referenced in the config statement. The found set of specifications contains the minimal needed (transitive) dependencies. This set of specifications, together with the check (or run) command and referenced assert and config statements make up the specialized check-module.

Applying Forget and Mock During preparation the execution of the **forget** and **mock** operators are also performed.^{††} Both **forget** and **mock** manipulate the specification dependency graph by removing those dependencies that are not needed any more after applying **forget** and **mock**. Applying **forget** results in a subgraph of the original dependency graph (with edges removed). Applying **mock** results in a graph that overlaps the original graph but also holds the newly inserted mocked specification. In both cases the reachability analysis is performed again after the alteration of the graph resulting in the minimal set of specifications that is needed to perform the check or run command. Figure 5.2 shows examples of applying **forget** and **mock** to the specification dependency graph.

Next to altering the dependency graph both operators also rewrite parts of the specification as well. *Forget* slices out the field that is to be forgotten from the specification. It removes both the field definition as all constraints with references to the field in the pre- and postconditions of the events. This is done via a data-flow analysis in which the use-definition relation is traversed for the fields that need to be forgotten.

Mock performs a *rename* on the specification that is configured as a mock. The mock is renamed to the original (the specification of which it is a mock). This renaming is necessary to make sure the new combination of specifications is well-formed again since the unaltered specifications still reference the original, non-mocked specification by name.

^{††}Appendix C.2 contains the implementation of the ‘forget’ and ‘mock’ algorithms.

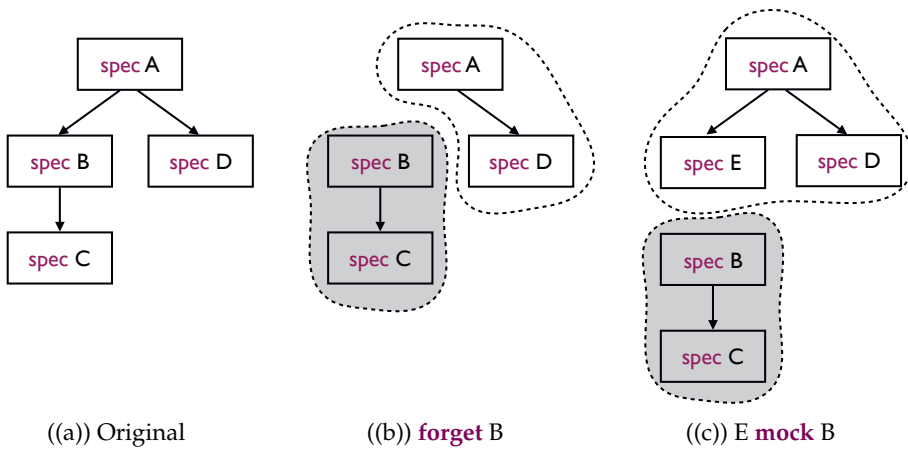


Figure 5.2: Example of an specification dependency graph (a) and the consequence of applying **forget** (b) and **mock** (c) operators. The specifications in the grayed subgraphs are removed.

5.4.2 Step 2: Normalization

Normalization of a module entails three parts: (1) Inlining the state definitions as local fields, (2) Adding frame conditions to the events and (3) Adding a frame event. The result of normalizing a REBEL2 specification is again a valid REBEL2 specification. Normalization is purely the application of local transformations. Figure 5.3 shows the effect of normalizing the (slightly altered) Transaction specification. We will discuss every part separately.

Inlining State Definitions As mentioned in the formalization section 5.3 the life cycle definition in a REBEL2 specification becomes part of the local fields. A simple transformation is performed on each specification that introduces a new field state with a new type representing the states as defined in the original specification. A new specification is added to represent the new state type. Listing 5.3(b) contains an example of such a new ‘state specification’ (line 18). The definition of such a specification also introduces *constant instances* (CREATED and BOOKED in our example). These constant instances are instances of the TState specification that are implicitly part of each config statement. They can be referenced as constants by other specifications (i.e., see line 14 or 16 in Listing 5.3(b)). Constant instances serve a similar purpose as enumeration types do in other languages.

Next to the addition of this new ‘State’-type, the pre- and postconditions of the events are strengthened with constraints on the newly added state field. These constraints simulate the life cycle definition of the original specification. Lines 14 and 16 contain the constraints to simulate the life cycle definition of the defined Transaction.


```

1 spec Transaction
2   frm: Account,
3   to: Account,
4   amt: Integer;
5
6   init event create(...) ...
7   final event archive() ...
8
9   event book()
10  pre:
11    this.frm.withdraw(this.amt),
12    this.to.deposit(this.amt);
13
14  states:
15    (*) -> created: create;
16    created -> booked: book;
17    booked -> (*): archive;

```

((a)) Original specification.

```

1 spec Transaction
2   frm: Account,
3   to: Account,
4   amt: Integer,
5   state: TState;
6
7   init event create(...) ...
8   final event archive() ...
9
10  event book()
11  pre:
12    this.frm.withdraw(this.amt),
13    this.to.deposit(this.amt),
14    this.state = TState[CREATED];
15  post:
16    this.state' = TState[BOOKED],
17    this.amt' = this.amt,
18    this.frm' = this.frm,
19    this.to' = this.to;
20
21 spec TState[CREATED,BOOKED];

```

((c)) Adding frame conditions.

```

1 spec Transaction
2   frm: Account,
3   to: Account,
4   amt: Integer,
5   state: TState;
6
7   init event create(...) ...
8   final event archive() ...
9
10  event book()
11  pre:
12    this.frm.withdraw(this.amt),
13    this.to.deposit(this.amt),
14    this.state = TState[CREATED];
15  post:
16    this.state' = TState[BOOKED];
17
18 spec TState[CREATED,BOOKED];

```

((b)) Inlining life cycle.

```

1 spec Transaction
2   frm: Account,
3   to: Account,
4   amt: Integer,
5   state: TState;
6
7   init event create(...) ...
8   final event archive() ...
9
10  event book()
11  pre:
12    this.frm.withdraw(this.amt),
13    this.to.deposit(this.amt),
14    this.state = TState[CREATED];
15  post:
16    this.state' = TState[BOOKED],
17    this.frm' = this.frm,
18    this.to' = this.to;
19
20  event frame()
21  post: this.amt' = this.amt,
22        this.frm' = this.frm,
23        this.to' = this.to,
24        this.state' = this.state;
25
26 spec TState[CREATED,BOOKED];

```

((d)) Adding framing event.

Figure 5.3: Result after each normalization step of the Transaction specification.

Adding frame conditions A known problem when modeling behavioral systems declaratively is that a user must not only state what changes, but also what *does not* change otherwise the system might be under-constrained and thus start to behave in an unexpected manner. To relieve the user of this extra burden REBEL2 offers a simple heuristic; those fields whose next value is not referenced in a postcondition will be automatically added as frame condition. Listing 5.3(c) shows an example of this. In the book event the `frm`, `to` and `amt` fields are not referenced in the the postcondition. During normalization constraints are added to frame the value of these fields in the next state.

This is also where the `init` and `final` modifiers come into play. Frame conditions are not added in events flagged with these modifiers since this would lead to unsatisfiable constraints (a variable can not have a value if the machine is not in an initialized state).

Please note that this heuristic is not conclusive. For instance when a fields next value is referenced in a postcondition but the formulated constraint is under-constrained allowing for multiple valuations. But considering that mistakes are often easily spotted during model checking and the fact that a user can always add a custom frame condition to the postcondition instead of relying on the automatic addition we feel that this is less of a problem.

Adding frame event REBEL2 allows for the model checking of multiple instances of specifications at the same time. In each step however, only one instance may make a step (or multiple if the step entails synchronization of events). The other instances must by definition keep their current values (see Formalization, Section 5.3). To facilitate this a local frame event is added to every specification. This event is raised whenever the instance is not (part of) the instance that makes a step. The frame event frames all field values in the next state to the values of the current state (see Listing 5.3(d)).

5.4.3 Step 3: Translation

After normalization the resulting specification(s) are translated to an ALLEALLE problem [SvdSV19].[‡] ALLEALLE is a *relational model finder* which searches for satisfying relational instances of a given relational problem.

Short introduction of ALLEALLE ALLEALLE's logic combines *relational algebra*, *first order logic* and *transitive closure*. ALLEALLE is similar to the relational model finder Kodkod [TJ07b]. The difference is that ALLEALLE utilizes a SMT solver instead of the SAT solver used by Kodkod. This means that ALLEALLE can utilize a direct encoding of constraints over integers and strings without needing a specialized

[‡]See Appendix C.4 for the complete translation algorithm as implemented in Rascal.

boolean encoding. To allow for direct constraints on (integer) attributes ALLEALLE uses relational algebra as introduced by Codd [Cod83] as its underlying logic which has the *selection* operator to express constraints on attributes directly.

An ALLEALLE problem contains two parts: i) Relational definitions and ii) Constraints on these relations. The definition of ALLEALLE relations come from the relational model [Dat94]. This definition prescribes that a relation contains a *header* and a *body*. The header defines the attribute names and domains, the body contains the (potential) tuples. ALLEALLE is bounded meaning that a relation can not hold more or different tuples than described in its upper bound. Next to an upper bound, a relation can have a lower bound. Tuples defined in the lower bound must always be present in the relation.

The constraints are formulated using a combination of relational algebra, first order logic and transitive closure. An ALLEALLE problem is satisfiable if there is a valuation of each relation for which both the tuple bounds and the constraints hold. There can be multiple satisfying instances possible. Finding these instances is done by the ALLEALLE model finder.

Encoding REBEL2 specifications as ALLEALLE problems ALLEALLE, like Kodkod, is a general purpose model finder. It does not offer built-in support for encoding transition systems. This needs to be encoded in the problem itself. To encode the transition system we use a similar encoding as described by Cunha [Cun14]. In essence it means that every value that can change between each step in the transition system is encoded as a ternary relation. For instance, the balance of an Account can change in each step of the transition system therefore the ALLEALLE relation representing this value is defined as the ternary relation: $\text{Configuration} \times \text{Account} \times \text{Integer}$. We use the term *Configuration* to describe the state of the LKS as a whole since the term ‘State’ is highly ambiguous.^{§§} The *Account* relation holds all the instances of the Account that are defined in the config statement.

Next to the *Configuration* relation there is the binary *step* relation which encodes the order of *Configuration*. The maximal number of *Configurations* and *Steps* depend on the max steps defined in the check or run command.

Field and event parameter multiplicity definitions (optional, set or scalar) are encoded as additional ALLEALLE constraints. Every event definition becomes a constraint which utilizes the *step* relation and encodes the fact that there can be only one event that is raised in every step. The LTL expressions of assumptions and assertions also utilize the *step* relation in their translations to ALLEALLE.

After the translation the translated problem is given to the ALLEALLE model finder together with the minimization criteria to find a solution in the least number of

^{§§}In each *Configuration* of the LKS every *instance* of a specification has its own current ‘State’.

steps. This means that in the case that a counter example exists (or witness, depending on the executed command) the model finder will return the shortest path.^{¶¶}

5.4.4 Step 4: Interpretation of the Result

The last step of the model checking pipeline is the interpretation of the result. If the ALLEALLE problem is not satisfiable the user is prompted with the message that ALLEALLE can not find a satisfying model. If this is the outcome of running a check command it might mean that the checked assertion holds but since the used model finding technique is bounded this is not guaranteed.^{***}

If the relational constraints of the generated ALLEALLE problem are satisfiable, the results are interpreted back into the domain of REBEL2 and presented to the user as an *interactive trace*. It enables the users to step through the found trace. The specified machines are visualized as UML Statecharts with data. The different instances are separately displayed as composite machines.

5.5 EVALUATION

We evaluate both the expressiveness of REBEL2 and the effectiveness of mocking for model checking by implementing two case studies, one from the automotive domain, and one from the financial domain.

5.5.1 Case Study – Exterior Lighting System

As part of the ABZ conference of 2020 the real-world case study “Adaptive Exterior Light and Speed Control System” (ELS and SCS respectively) was presented [HR20]. We have implemented a part of the case study, consisting of a model of the direction indicators and hazard warning lights system, to compare the expressiveness, conciseness, and overall usability of REBEL2 with others state-based implementations.

The system is split into three different subsystems: 1) Input 2) Sensors and 3) Actuators.^{†††} The case also prescribes a timing component: the direction lights must blink 60 times per minute. This means that, when blinking, a full cycle (from bright to dark) must be completed every second. Like other methods, REBEL2 does not support continuous time but it is possible to model a Timer machine to simulate time at every step. This Timer has a single invariant: time always flows forward with each successive behavioral step of the system. The Timer can be used to (partially) model the timing requirements stated in the requirements.

^{¶¶}This is a usability feature. shorter paths are easier to explore and comprehend when a bug is found.

^{***}There still could potentially be a counterexample if the bounds would be extended.

^{†††}See <https://github.com/cwi-swat/rebel2/tree/master/examples/paper/els> for the encoding of the ELS case study in REBEL2.

Table 5.1: SLOC comparison between different methods.

Method	SLOC	Included files
ASMeta [ABG ⁺ 20]	361	CarSystem001
Event-B [MFL20]	455	M2 (not plain text, only lighting related lines)
Classical-B [LMW20]	363	Sensors, PitmanController_v6, PitmanController_TIME_v4, GenericTimers, BlinkLamps_v3
Electrum [CML20]	155	AdaptiveExteriorLight_EU (only lighting related lines)
REBEL2	244	Actuators, Input, Sensors, Timer

Table 5.1 contains an overview of the different implementations in terms of Source Lines of Code (SLOC), restricted to the part that we have implemented. As can be seen in Table 5.1 the REBEL2 specification is comparable with the other implementations in terms of size, sitting between the Electrum and ASM / Classical-B implementations.

Out of the 13 stated requirements for the direction indicators and hazard warning lights the REBEL2 specification covers 11. The two missing or impartial requirements (ELS-4 and ELS-6) are concerned with modeling variants of the lighting system for different markets (e.g., EU versus USA) and the timing of the blink cycle when switching from tip-blinking to continuous blinking. Table 5.2 provides an overview of the fulfillment of the requirements of each implementation, as extracted from cited papers and available code.

We specified 17 assertions to check relevant properties of direction indicators and the hazard warning light. The initial decomposition of the system into four separate specifications (Actuators, Sensors, Input, Timer) facilitated checking local properties of each specification. The full sensor values contain more information than is needed to check the behavior of the direction and warning lights, so they could be mocked out. The mock state machine of Sensor only needed to specify single value (whether or not the key was in the ignition on position) to fully support the assertions, which resulted in a model checking speedup of approximately 1.17x.

The case description also documented scenarios describing the input and expected output values of all subsystems at a given time. For instance, the direction indicator scenario contains 26 steps. This scenario was represented as a dedicated (linear) state machine with 26 states, where each transition encoded a step of the scenario. The events corresponding to the steps capture the given sensor values and inputs as preconditions and the expected output values as postconditions.

Table 5.2: Fulfillment of direction blinking and hazard warning lights requirements.

Method	D.B. ¹	H.W.L. ²	Remarks
ASMeta [ABG ⁺ 20]	●	●	No time management, simulated via events. Blinking frequency is missing.
Event-B [MFL20]	●	●	Blinking frequency is missing.
Classical-B [LMW20]	●	●	Presented solution only addresses directional blinking and hazard warning lights.
Electrum [CML20]	◐	●	No time management. All integer values are replaced by enumerations.
REBEL2	●	●	Variants (EU-USA) not modelled. Time management partially implemented.

1) Direction Blinking (ELS 1–7) 2) Hazard Warning Lights (ELS 8–13)

● = Fully implemented

◐ = Fully implemented with minor omissions

◑ = Implemented but with omissions.

5.5.2 Case Study – Debit Card Lifecycle

This case study stems from the direct collaboration between the authors and a large bank, and describes the lifecycle of debit cards for payments or ATM withdrawals.^{###} The case study involves three key REBEL2 specifications: DebitCard (97 SLOC), Limit (weekly withdraw limits; 45 SLOC), and Date (full date specification, including leap years, day of the year, and day of the week; 102 SLOC).

The specified assertions either check for desired behavior (e.g., “Can a debit card be produced and activated?”) or check a safety property (e.g., “A customer should not be able to use a debit card after three failed PIN code attempts”). Per specification the model checker was ran on a configuration without mocked specifications and one with mocked specifications, for a total of 9 different assertions. The benchmark was run on a MacBook Pro (late 2015 model) with an Intel i5 processor and 8GB of RAM using Java version 11 (AdoptOpenJDK, build 2018-09-25), Rascal version 0.18.2 and Z3 version 4.8.8.

Table 5.3 shows the results of the experiment. As can be seen, all the checks with mocked specifications complete faster than their counterparts without mocked

^{###}See <https://github.com/cwi-swat/rebel2/tree/master/examples/paper/debitcard> for the REBEL2 encoding of the Debit Card case.

specifications, with an overall speedup factor in the range of 2x–5x. Most phases of the checking pipeline are faster with mocked specifications, except the preparation phase. The phase that mostly benefits of mocking is the solving phase. The speedup factor for this phase is between 2x to 234x (not taking the checks that timed out into consideration). Four checks (“CanAddOverrideAndCheck”, “CanOverdrawLimit”, “DebitCardCanBeProduced” and “CardCanExpire”) could not be checked with the configuration without mocked specifications, due to time-outs. Their counterparts with mocked specifications could however be checked within reasonable time.

As a proxy of the size of the explored state space, we report on the number of declared SMT variables and SMT clauses. Table 5.3 shows a decrease in the case of most mocked specifications, suggesting that solving time goes down because the solver has to perform fewer valuations. Nevertheless, “BlockedAfterThreeAttempts” and “Max3WrongPinAttempts” can still be solved while requiring more SMT variables and clauses than “DebitCardCanBeProduced” and “CardCanExpire”, which result in a time-out.

5.5.3 Discussion

We found that REBEL2 is expressive enough to implement both case studies with the exception of the continuous time aspect of the automotive case. This problem is not specific to the REBEL2 language as other formalisms struggled with this same issue.

Next to the expressiveness of the language we evaluated the effectiveness of mocking for model checking. We found that, especially in the financial case study, mocking allowed for the checking of properties that resulted in time-outs without mocked specifications. This however comes at an expense: using **forget** and **mock** makes model checking inherently unsound. In other words, it is possible to validate a property of interest in isolation, which will not hold when the system is considered as a whole.

The unsoundness of **forget** and **mock** may seem dangerous from the point of formal correctness, but lack of soundness is accepted in many other areas of validation and verification. For instance, the “small scope”-hypothesis (most bugs are found in a small scope) is at the heart of light-weight formal methods as, for instance, promoted by Alloy; nevertheless it is technically unsound. Similarly, in bug-finding and static analysis, it is well-known that many of these analyses are inherently unsound [LSS⁺15], but that does not diminish their usefulness. Finally, mocked specification are similar to mocked objects used for testing in object-oriented software development [MFC00]. Mocked objects often have different behaviors than the actual objects they substitute for, but the benefit of using them is widely acknowledged [SAB⁺17].

The **forget** and **mock** operators in REBEL2 are designed to support a more flexible, conversational style of checking properties, much like unit testing or property-based

Table 5.3: Comparison between model checking with and without mocking for the Debit Card case. Reported times are the found median in seconds after 10 runs.

	Without mocking						With mocking								
	Prep. (sec.)	Norm. (sec.)	Trans. (sec.)	Solve (sec.)	Total (sec.)	#vars (SMT)	#clauses (SMT)	Prep. (sec.)	Norm. (sec.)	Trans. (sec.)	Solve (sec.)	Total (sec.)	#vars (SMT)	#clauses (SMT)	
<i>Limit</i>															
- CanInitializeLimit	4.40	4.50	9.20	23.40	41.50	314.00	715708.00	1.80	1.80	5.50	0.10	9.10	222.00	348399.00	
- CanAddOverrideAndCheck	4.20	4.80	23.10	t/o	t/o	732.00	2144348.00	1.80	1.70	13.80	0.70	17.90	504.00	1045782.00	
- CantOverdrawLimit	4.60	4.80	40.90	t/o	t/o	1150.00	3587839.00	1.80	1.70	22.90	2.70	29.10	786.00	1758738.00	
- LimitsAlwaysPositive	4.20	5.10	33.30	2.10	44.60	941.00	2858968.00	1.80	1.70	18.40	1.20	23.10	645.00	1394935.00	
- AlwaysInSameCurrency	4.10	4.80	33.80	1.70	44.50	941.00	2858936.00	1.80	1.70	18.40	1.00	22.90	645.00	1394903.00	
<i>DebitCard</i>															
- DebitCardCanBeProduced	6.40	10.30	68.30	t/o	t/o	1471.00	8557656.00	8.50	3.30	23.90	0.60	36.30	883.00	641575.00	
- CardCanExpire	7.00	10.30	120.20	t/o	t/o	2337.00	14258727.00	8.60	3.20	40.90	1.90	54.70	1405.00	1069529.00	
- BlockedAfter3Attempts	5.80	7.70	259.20	33.90	306.60	4502.00	28510441.00	9.60	3.40	88.10	3.80	104.90	2710.00	2137854.00	
- Max3WrongPinAttempts	5.90	7.90	264.80	36.50	315.20	4502.00	28510471.00	9.30	3.40	89.70	3.90	106.30	2710.00	2137884.00	

t/o = Timed out after 10 minutes.

testing in software development [Runo6]. One of way of stating this is: REBEL2 favors timely feedback over logical soundness.

Another added benefit of **mock** and **forget** is that having language constructs specific for abstraction helps making the applied abstractions needed for model checking explicit where they would otherwise remain implicit for the unsuspecting reader. In other words, a specification language without these constructs expects the user to create abstract specifications in the first place, rendering it difficult for readers to see which properties were abstracted from.

5.6 RELATED WORK

Alloy is a popular lightweight formal specification language based on relational logic with transitive closure [Jac12; Jaco2b]. Alloy allows for bounded model finding by translating specifications to SAT formulas and utilizing an external SAT solver [TJ07b]. Like REBEL2, the user specifies the bounds of a problem, which is used during model finding. Because of Alloy's generality specifying behavioral problems (which require some sort of transition system) requires an encoding in the relational logic of Alloy.

Electrum extends Alloy with temporal operators [MBC⁺16] to make such encodings more direct. The temporal operators can be used to express safety and liveness properties and operate over so called *variable* relations, relations whose contents can change over time. DynAlloy [FGL⁺05; RCG⁺17] is a dynamic logic-based extension of Alloy, supporting partial correctness reasoning via actions and action composition. The addition of actions in DynAlloy obviates the need of an explicit encoding of the transition system, but it does not offer support for LTL formulas, which makes expressing liveness properties hard.

Both Electrum and DynAlloy can be used to model structural and behavioral problems, but differ from REBEL2 in a number of ways. First, since Alloy translates specifications to SAT formulas it is hard to reason about non-relational data, such as integers, reals, or strings. Since REBEL2 translates it specifications to ALLEALLE which in turn utilizes an SMT solver, REBEL2 offers native reasoning support in the theories supported by the solver. Second, Alloy, Electrum and DynAlloy support modularizing specifications using modules and inheritance but lack the forget and mock mechanisms of REBEL2. As a result, feasibility of checking properties is relative to the full specification, rather than the property of interest. We do believe however that both constructs could be implemented in these formalisms.

Abstraction is a key mechanism to control for complexity. In the context of formal specification this applies to both a complexity reduction for humans, as well as a potential reduction in the search space for automated proofs and model checking. For instance, the specification language mCRL2 [GM14] offers the primitives *internal action* or τ -step and the *abstraction operator* (τ_I) for this purpose. Another approach,

which lies at the heart of formalisms such as Event-B [AH07] and ASM [Bör98], is the concept of refinement. The specifier starts with a high-level specification of the system which is then gradually refined into more detailed specifications. Each refinement step must be proven to be a correct via proof obligations which can have to be discharged, either using automatic tool support, or manually by providing a proof. REBEL2's `mock` and `forget` can be seen as similar operators to eliminate detail from a specification, in order to make model checking more feasible.

5.7 CONCLUSION

In this paper we have introduced the specification language REBEL2 which contains two language constructs, `forget` and `mock`, that offer the possibility to apply mocking during model checking. These two constructs allow the user to reduce the state space which needs to be traversed by the model checker. They can be used to redefine (parts of) the specification during model checking without having to change the original specifications. Users can specify the problem at hand without worrying about the impact on model checking at design time, but rather defer such concerns until actually checking a property of interest. We conjecture that this makes REBEL2 suitable for specifying industry-scale systems, such as those found in large enterprises, while still being able to verify (parts of) a system using model checking.

We have evaluated REBEL2's expressiveness and the effectiveness of model checking with mocking by implementing two industrial case studies. In the first case study – originating from automotive domain – we compared REBEL2 with existing solutions in alternative frameworks (ASMeta, Electrum, Classical-B and Event-B). The results showed that REBEL2 can be used to specify such problems in roughly the same number of lines code, and that applying the `mock` construct to parts of the specifications sped up model checking by a factor of roughly 1.17x. In the second case study we investigated the effectiveness of model checking with and without mocking. This case stemmed from the financial industry and was conducted together with employees from a large bank and showed that applying mocks while model checking improved the overall checking times up to 5 times. In some cases it made checking possible where performing the model checking without mocking resulted in time-outs of the underlying solver.

Further research directions include 1) extend REBEL2 to support deadlock detection, since this is now an impediment to model checking; 2) implement different slicing algorithms (e.g., [EHP⁺18]) and assess the impact in terms of performance and soundness; and 3) provide empirical corroboration of the mocking hypothesis.

To summarize, REBEL2 is a formal specification language aimed at large industrial enterprise settings, which brings the concept of mocking to the world of formal methods, providing faster model checking feedback when checking behavioral properties of interest.

DESIGN & IMPLEMENTATION

In the previous chapters we have discussed ideas and methods that we have developed in our search for lightweight specification and verification techniques for Enterprise Software such as found in banks. In this chapter we will zoom in on the different software components that we have created. We will discuss the design decisions that were made and highlight some of the trade offs between different solutions.

Figure 6.1 shows a general overview of the different software components that we developed. All components were developed in Rascal [KvdSV09]. We will discuss each component separately in the coming sections.

6.1 REBEL, VERSION 1

REBEL was our first version of a specification language for a bank.* Chapter 2 contains an overview of the language but we will list its key elements here for completeness:

- Easy to understand formalism based on Extended Finite State Machines
- Domain specific types such as **Money**, **Currency** and **IBAN** for natural encoding of problems in the financial domain.
- Designed with Product Line Engineering in mind by separating declaration and definition of events and invariants to increase reusability.
- Built-in simulator that allows users to step through a single state machine
- Built-in reachability verification

This first version of Rebel contained the following parts:

- Syntax definition of the language.
- A type checker.
- A module system.
- Web-based visualizations of specifications.
- A translation from the REBEL language to SMT-LIB.
- A simulator utilizing the Z3 SMT solver.
- A model checker utilizing the Z3 SMT solver.

In total REBEL consists of approximately 4000 lines of Rascal code of which the majority of code is part of the Model Checker with 1700 lines of Rascal.

*<https://github.com/cwi-swath/rebel>

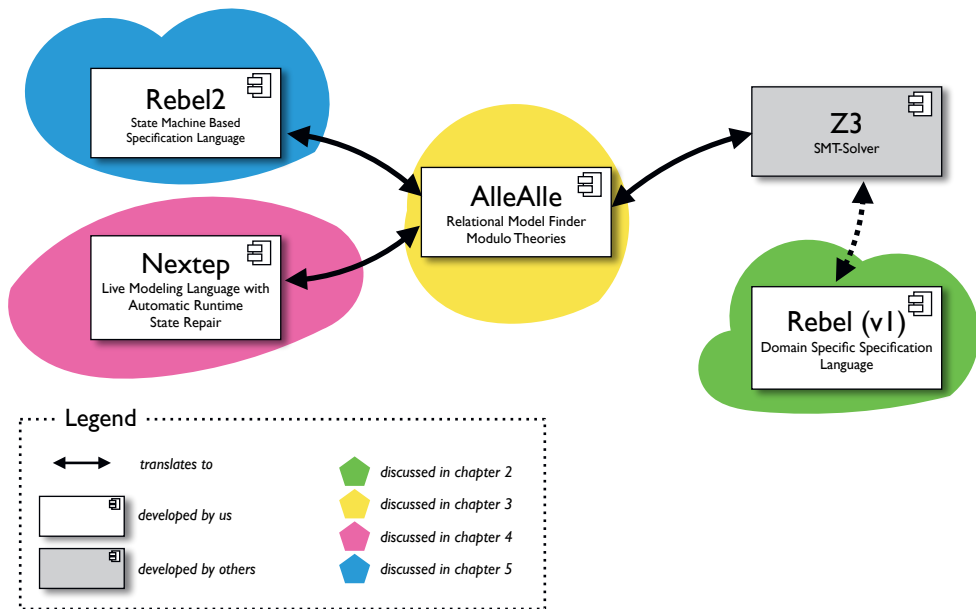


Figure 6.1: Overview of the different software components developed for the performed research described in this thesis.

6.1.1 Postmortem analysis - Lessons learned

Although the first version of REBEL contained many features, it was not complete. This had to do with a number of reasons of which we will list three of the most prominent:

Incomplete Symbolic Compiler During the development of this first version of REBEL we experienced that constructing a symbolic compiler for a language requires a tremendous engineering effort (as others before us [TB14]). Our symbolic compiler worked by mapping each REBEL language construct onto a formula in SMT. Creating a correct, complete, reusable and maintainable mapping turned out to be a difficult task. We found that the main reason was that the semantic distance between the input language (in this case the REBEL specification language) and the the target language (SMT-LIB) was (too) large. As a result we were unable to create a mapping that was complete for all constructs of the input language.

Inflexibility of the type system We specifically designed REBEL to contain domain specific types such as **Money**, **IBAN** and **Currency**. Although these domain specific

types made that the specifications were easy to read it also made extending the language with other types hard. Each type, including its operations, needed to be implemented as a primitive in the language. Next to that, each type and its operations needed a specific translation to SMT. This combination made the language inflexible for change. This hampered the applicability of the language since we needed to change the language when modeling problems from an adjacent financial domain (for instance, modeling bank accounts required different types than modeling mortgages).

Inflexibility of the specification language Originally the tooling allowed for the checking of invariants for a specified bound. The chosen syntax to define these properties was fairly limited compared to properties that can be defined using Linear Temporal Logic (LTL) or Computational Temporal Logic (CTL) which made it hard (or even impossible) to express interesting and deep properties. This in turn limited the applicability of the model checker.

6.2 ALLEALLE

We created ALLEALLE to bridge the earlier described semantic gap between the input language, REBEL, and the target language SMT-LIB.[†] The creation of ALLEALLE was inspired on the ALLOY language and its relational model finder KODKOD [Jac02b; TJo7b].

6.2.1 Background of Alloy and Kodkod in relation to AlleAlle

ALLOY was created by Jackson in the early 2000's as a lightweight specification language with an emphasis on partiality (partiality of models, partiality of language, partiality of analysis). It uses Tarski's relational algebra as underlying formalism. Everything you can express in ALLOY is seen as an relation of different arities (objects are expressed as unary relations, fields on these objects as binary relations, etcetera) and constraints over these relations using first order logic and relational algebra [Jac12]. KODKOD is the relational model finder used by ALLOY to translate ALLOY specifications to SAT formula's. KODKOD extends the language of ALLOY with the definitions of relational bounds. These bounds contain the minimal number of tuples (lower bound) and maximal number of tuples (upper bound) that a relation can contain. Since the analysis performed by KODKOD is bounded every defined relation must have an upper bound. KODKOD's main contribution is its highly efficient translation of the relational input language to SAT formulas. An existing SAT solver is used to solve KODKOD's translated SAT formula. Found valuations are then translated back to the level of the relational input language.

[†]<https://github.com/cwi-swat/allealle>

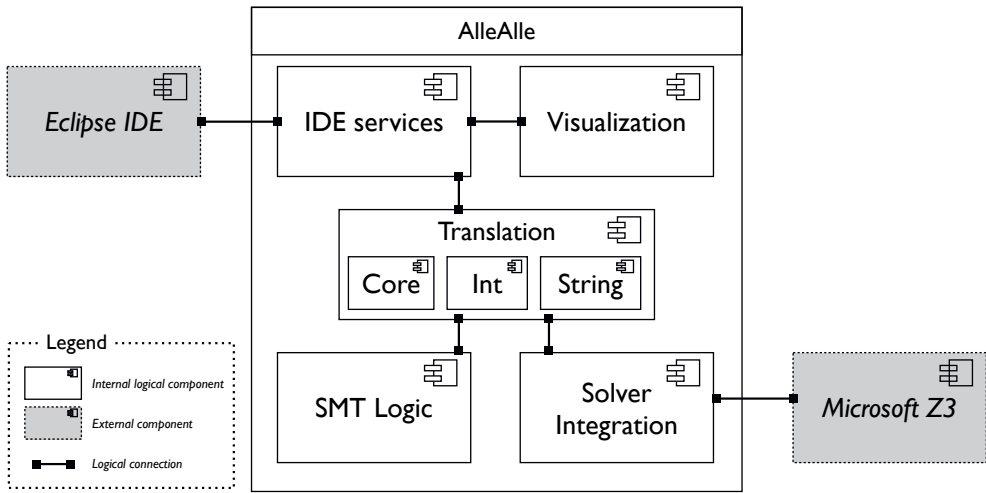


Figure 6.2: Overview of the different components of ALLEALLE.

This decoupling of user-facing specification language, the ALLOY language, and an intermediate language, KODKOD, that sits between the host language and the target language (in level of abstraction) as well as the translation scheme of KODKOD to a SAT-formula, was inspiration for the research and development of ALLEALLE. The reason that KODKOD itself was less suitable for our work was that it is not optimized for reasoning on other theories than relational logic. Although it is possible to define integer constraints and use integer operations, translating and solving these constraints are not ideal since everything is translated to SAT formulas. As one can imagine our problem domain, financial systems, often contain integer arithmetic constraints. Using KODKOD and SAT solvers as back-end formalism is because of this sub-optimal.

6.2.2 Components of ALLEALLE

Figure 6.2 shows an overview of the different components of AlleAlle. We will briefly describe each component:

- IDE Services - Registers the ALLEALLE language and other IDE support (such as type checking and outlines) with the Eclipse IDE.
- Interactive Visualizations - Offers visualizations of the found ALLEALLE models. Models can be displayed as (augmented) graphs or as tables. Allows the user to step through the different found models (via a 'next model' button).
- Translation - Contains the core functionality of ALLEALLE. Contains the translation algorithms to translate an ALLEALLE specification to an SMT

Table 6.1: Overview of Source Lines Of Code (SLOC) for the different components of ALLEALLE. Source lines of tests are not counted.

Component	SLOC
IDE Services	531
Visualizations	262
Translation	
- <i>Core</i>	1594
- <i>Int</i>	389
- <i>String</i>	143
SMT Logic	164
Solver Integration	170
Total	3253

formula. Also contains the concrete and abstract syntax definitions of the ALLEALLE specification language.

- SMT Logic - Is used by the translation component. Contains an Abstract Syntax Tree (AST) definition of allowed constructs in `SMT-LIB`.
- Solver Integration - Handles the communication to and from the external SMT solver. Streams the translated SMT formula to the solver and reads the result.

In total ALLEALLE is made up of 3253 lines of source code. Table 6.1 contains a breakdown of the size of the different components.

6.2.3 The translation algorithm

The core of ALLEALLE is its translation algorithm[‡] that translates an ALLEALLE specification to an SMT formula. Figure 6.3 shows an overview of the complete translation flow from specification to relational model. We will discuss each separate step according to this flow.

The first step in the translation is the conversion of the relational definitions to the internal data representation as described in section 3.4.2. The second step uses these internal relational representations to perform the actual translation to an SMT formula as earlier described in 3.4.2. This SMT formula is internally represented as an AST. The translation is implemented using a recursive descent traversal style since each construct can be converted without the need of extra information from the context (context free).

[‡]As described in Chapter 3, Section 3.4.2.

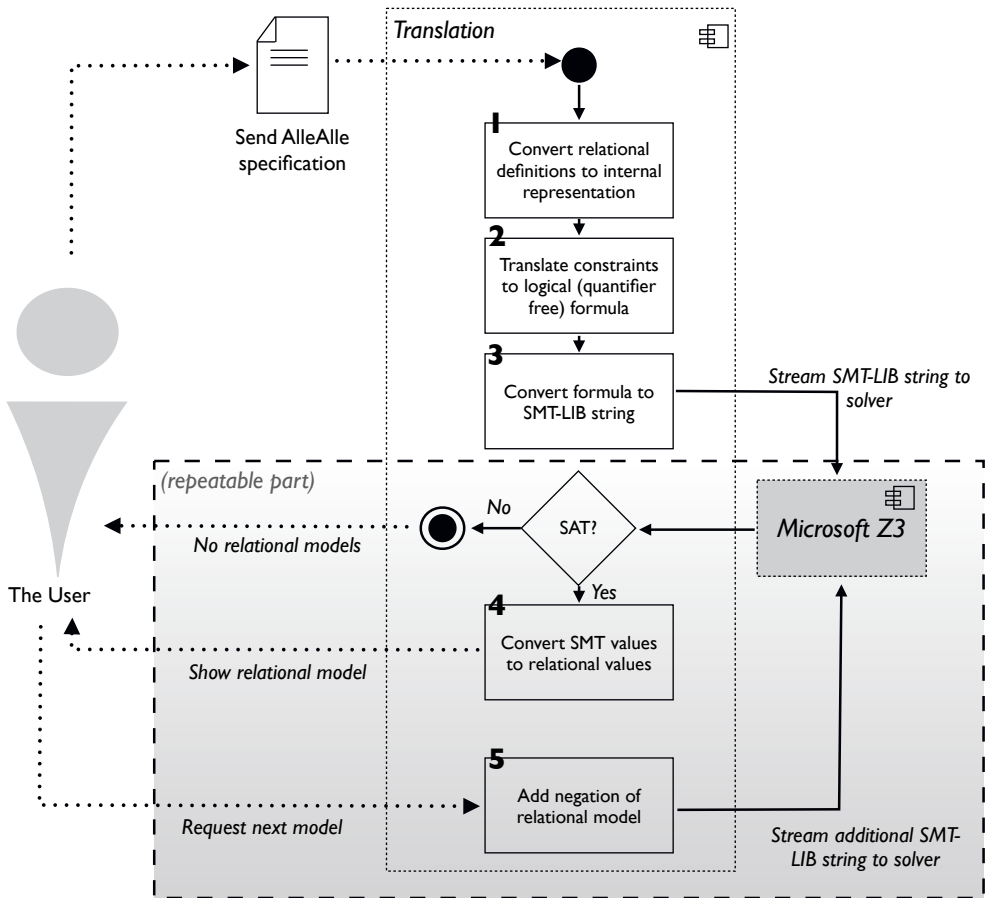


Figure 6.3: From ALLEALLE specification to relational model.

The result of the translation is an AST which needs to be converted to SMT-LIB *assertions* (step 3). Before this assertion can be interpreted by the SMT solver the variables, which are referenced in the formula, must be declared in SMT-LIB (using the construct `declare-const`). The type (or sort) of the variable depends on its definition in the ALLEALLE specification (`bool`, `int`, `string`). Next the formula is constructed in SMT-LIB. The whole formula is formulated as one `assert` statement. The root of this formula is a conjunction with many nested disjunctions and conjunctions. No normalization is performed during this phase. If the original ALLEALLE specification contains optimization criteria, then these criteria are translated to a `minimize` or `maximize` commands. The last statement that is streamed to the SMT solver is the

`check-sat` command which instructs the solver to solve the assertion. Depending of the configuration, the solver can return four different answers:

- The formula is *satisfiable*.
- The formula is *unsatisfiable*.
- The solver reached a configured *time-out*.
- The solver cannot prove or disprove the assertion and returns *unknown*[§]

In the last three resulting cases (unsatisfiable, time-out or unknown satisfiability) the user is prompted that no model was found and is shown the result from the solver. If the assertion is satisfiable, the solver assigned variable values are queried (using the `SMT-LIB` `get-values` command). These values are mapped back to the level of the original relational problem resulting in a relational model in which all relations have tuples assigned (or are empty) and each integer and string attribute in a tuple has a value assigned (step 4).

Whenever there is a model returned by the solver the user can request the solver to return the *next model*. To get the next model the current model is mapped back to `SMT-LIB` variables and an assertion is constructed negating the current valuations of the variables (step 5). This new assertion is streamed to the solver. Since the solver and its context (the solver process) are left alive after the initial solve, this new assertion is added to the original assertion. The solver will again try to solve the original assertion which now needs to result in different value assignments. If the solver can find another assignment of values this again is queried and mapped back to the relation model and presented to the user. These last steps (finding a next model) can be repeated until the solver can not find new valuations anymore.

Algorithmic complexity of the translation The time it takes to perform a translation is mostly dependent on the number of nested quantifiers defined in the `ALLEALLE` specification (see also Figure 3.8 in Chapter 3). The time grows exponentially depending on the number of nested quantifiers. The algorithmic time complexity of the translation algorithm is therefore in $O(2^n)$ where n is the number of nested quantifiers in an `ALLEALLE` problem. Specifications with fewer nested quantifiers will have significantly shorter translation times. Some optimizations are implemented to increase the performance of the translation of quantifiers such as short-circuit boolean conjunction and disjunction detection and memoization of translated expressions and formulas. Other techniques that could improve the translation time such as skolemization[¶] are not part of our current implementation of `ALLEALLE`.

[§]`Z3` returns *unknown* if a formula is beyond its current reasoning power. Since theories can be undecidable, such as the theory of strings [BTV09], it is fairly common to run into the problem that the solver returns *unknown*.

[¶]Skolemization is a common technique to eliminate (nested) existential quantifiers by replacing them with functions.

```

1  (declare-const ParamEventPegAddD_c2_c3_d1 Bool)
2  (declare-const ParamEventPegAddD_c1_c2_d1 Bool)
3  (declare-const ParamEventPegAddD_c7_c8_d1 Bool)
...
367 (assert (and (or (not true) (= count_ (+ (ite (and Config_c2 true)
...
31161 (or (not ParamEventTowerMoveFrom_c5_c6_p1) order_c5_c6 )
31162 (or (not ParamEventTowerMoveFrom_c7_c8_p1) order_c7_c8 )
31163 (or (not ParamEventTowerMoveFrom_c4_c5_p1) order_c4_c5 )
31164 (or (not ParamEventTowerMoveFrom_c6_c7_p1) order_c6_c7 )
31165 (or (not ParamEventTowerMoveFrom_c1_c2_p1) order_c1_c2 )
31166 (or (not ParamEventTowerMoveFrom_c3_c4_p1) order_c3_c4 )
31167 ))))
31168 (set-option :opt.priority lex)
31169 (minimize count_)
31170
31171 (check-sat)

```

Listing 6.1: Excerpt of the resulting SMT-LIB formula after translating the ALLEALLE specification for the Towers of Hanoi with three discs.

6.2.4 Design decisions

In the coming paragraphs we will discuss two important design principles of ALLEALLE: theory extensibility and term rewriting.

Theory Extensibility We designed ALLEALLE in such a way that it is open to extension of additional SMT background theory. The base translation algorithm of ALLEALLE translates the relational operators (this is depicted by the ‘core’ component in figure 6.2). The different background theories translations are used whenever there are constraints formulated on fields which are of a type other than `id`. These constraints can happen in two different syntactical constructs: in the restriction operator (the `where` construct) or in an aggregation function (e.g., the `min` or `avg` aggregation functions are part of the `int` domain). In order to integrate new SMT theories with ALLEALLE new translation rules must be added for the specific restriction operations and, if applicable, for new aggregation functions. The amount of code needed to implement these new theories are, in our experience, limited (depending on the number of attribute operations required). For instance, implementing the Integer SMT theory required 386 source lines of code (see Table 6.1).

Term rewriting on translation When translating the ALLEALLE constraints to formulas in SMT-LIB normalizations can be applied. This is done by leveraging

```

1 // Definition of the SMT Formula AST
2 data Formula = \true()
3             | \false()
4             | pvar(str name)
5             | \not(Formula f)
6             | \and(set[Formula] fs)
7             | \or(set[Formula] fs)
8             ...
9
10 // Rewrite rules that are applied on construction
11 Formula \or({}) = \false();
12 Formula \or({Formula x}) = x;
13 Formula \or({\false(), *Formula r}) = \or(r);
14 Formula \or({\true(), *Formula _}) = \true();
15 Formula \or({*Formula a, \or(set[Formula] b)}) = \or(a + b);
16 ...

```

Listing 6.2: SMT Formula AST definition and the implemented constructor rewrite rules using Rascal. the * is the splice operator matching zero or more elements.

Table 6.2: Implemented ‘Level 1’ rewrite rules which are applied on construction of the SMT formula. ϕ denotes an SMT formula.

Operator	Rule	Result
or	$\top \vee \phi \vee \dots \rightarrow \top$	
	$\perp \vee \phi \vee \dots \rightarrow \phi \vee \dots$	
and	$\top \wedge \phi \vee \dots \rightarrow \phi \wedge \dots$	
	$\perp \wedge \phi \vee \dots \rightarrow \perp$	

Rascal’s built-in pattern matching mechanism on constructing AST’s. Whenever a new SMT-LIB formula is created Rascal applies pattern matching to match the constructor. Using constructor overrides, term rewriting can be applied so such that the SMT-LIB formulas are rewritten to a normal form on creation of the AST. Listing 6.2 shows how this can be implemented in Rascal.

Rascal pattern matching facilities are rich. Because of this powerful rewrite rules can be defined in a concise manner. For instance, if we would like to define the following rewrite rule: $\phi \vee (\phi \wedge \dots) \rightarrow \phi$ (were the \dots denote one or more terms of the conjunction) we can do this as follows:

```
Formula or(Formula a, and({a, *Formula _})) = a;
```

The * denotes Rascal’s *splice* operator matching zero of more elements.

Table 6.3: Implemented ‘Level 2’ rewrite rules which are applied on construction of the SMT formula. ϕ and ψ denote SMT formulas.

Operator	Rule	Result
or	$\neg\phi \vee \phi$	$\rightarrow \top$
	$\phi \vee (\psi \vee \dots)$	$\rightarrow \phi \vee \psi \vee \dots$
	$\phi \vee (\phi \wedge \psi \wedge \dots)$	$\rightarrow \phi$
	$(\phi \wedge \psi) \vee (\phi \wedge \psi)$	$\rightarrow \phi \wedge \psi$
and	$\neg\phi \wedge \phi$	$\rightarrow \perp$
	$\phi \wedge (\psi \wedge \dots)$	$\rightarrow \phi \wedge \psi \wedge \dots$
	$\phi \wedge (\phi \vee \psi \vee \dots)$	$\rightarrow \phi$
	$(\phi \vee \psi) \wedge (\phi \vee \psi)$	$\rightarrow \phi \vee \psi$
negation	$\neg\neg\psi$	$\rightarrow \psi$
	$\neg\top$	$\rightarrow \perp$
	$\neg\perp$	$\rightarrow \top$
implication	$\top \implies \phi$	$\rightarrow \phi$
	$\perp \implies \phi$	$\rightarrow \top$
	$\phi \implies \top$	$\rightarrow \top$
	$\phi \implies \perp$	$\rightarrow \phi$
	$\phi \implies \phi$	$\rightarrow \phi$
equivalence	$\phi \iff \phi$	$\rightarrow \top$
if-then-else	$\text{ite}(\top, \phi, \psi)$	$\rightarrow \phi$
	$\text{ite}(\perp, \phi, \psi)$	$\rightarrow \psi$
	$\text{ite}(\psi, \phi, \phi)$	$\rightarrow \phi$

Although these rewrite rules are very concise and effective they can have a negative impact on the translation time. For instance, matching elements in sets (i.e., ACI matching) has long been proven to be an NP-complete problem [Eke02; KN86]. As one can imagine, depending on the size of the original set, matching brings run-time overhead. While performing the overall translation these rules are constantly checked stacking up this run-time overhead. Not performing any rewrites on construction could however have a negative impact on the solving time since the solver (Z_3 in our case) needs to process a formula with more terms. This could, in theory, result in longer solving times since the SMT solver now needs to do the heavy lifting that was postponed during construction.

Term rewriting experiment

To get insight into the impact of term rewriting on translation we tested the translation and solving of formulas on a benchmark of ALLEALLE specifications. We constructed three different sets of rewrite rules:

1. No rewriting - Only empty clauses are removed.
2. Level 1 rewriting - Short circuiting is performed for dis- and conjunctions. See Table 6.2 for an overview.
3. Level 2 rewriting - Deeper patterns are matched and rewritten. See Table 6.3 for an overview.

We ran each set of rewrite rules on four different specifications in different configurations (increasing the upper bounds of the relations). Two of these specifications, FileSystem and Halmos Handshaking, were written directly as ALLEALLE specifications. The two other specifications, DebitCard and The Tower of Hanoi, were written in REBEL2. Since REBEL2 uses ALLEALLE as an intermediate language, the translation was captured and used in this benchmark. All experiments were performed on an Intel i7 CPU (10610U) containing 32GB of RAM running Ubuntu 20.04 with Rascal version 0.18.3 (stable), OpenJDK 11, Z3 4.8.13 and Eclipse 2020-06. Each experiment was ran 30 times (preceded by a 10 time warm-up). The reported times are the means of these 30 runs. Table 6.4 shows the result of the experiment.

The results show that applying rewriting on construction indeed gives a run-time performance overhead. A slightly counter intuitive result is that for some cases applying level 2 rewriting increases the performance of the translation. For instance, the translation of the Halmos specification is slightly quicker when translated using the level 2 rewrite rules instead of only the level 1 rewrite rules. A possible explanation for this could be that by applying more rigorous rewriting the number of terms in each AST node decreases. As such, the number of elements that need to be matched during the application of the rewrite rules go down. This is visible in the number of clauses that are left in the resulting SMT formula after rewriting. They are decreased significantly as a result of rewriting.

Looking at the reported solving times tells a different story. The impact of the more concise, normalized formulas is low. Even more counter-intuitive is that in some cases the smaller formulas have a negative impact on the solving times. For instance, the reported solving times for the ‘DebitCard - in 5 steps’ problem is quickest for the non-normalized formula. The reported Level 2 rewriting increases the solve time by a factor of 1,4x while the Level 1 rewrite rules increase the solving times with a factor of 1,7x even though in both cases the number of clauses decrease considerably. This experiment again shows that it is hard to predict the performance behavior of the solver given a formula. SMT solvers contain many heuristics to increase the speed in which they can find an answer [DP13]. Predicting which heuristic is used given a

certain problem without knowing the exact internals of each heuristic is a very hard problem.

Given the fact that the current version of ALLEALLE is often most effected by longer translation times and less by the solving times, we conclude that in the current version of ALLEALLE no rewriting on construction is performed.

Table 6.4: Comparison of different levels of formula rewriting during translation. Reported times are means of 30 runs (preceded by 10 warmup runs). Reported factor is compared to the 'No rewriting' column. Highlighted row and cells are discussed in the text.

Problem	No rewriting			Level 1 rewriting			Level 2 rewriting						
	Trans. (in ms)	Solve (in ms)	# Clauses	Trans. (in ms)	Solve (in ms)	Solve (factor)	Trans. (factor)	Solve (in ms)	Solve (factor)	Trans. (factor)	Solve (in ms)	Solve (factor)	# Clauses
DebitCard - in 3 steps	6364.00	550.00	2 004 313.00	8349.00	x 1.31	360.00	x 0.65	842 288.00	8916.00	x 1.40	430.00	x 1.19	641 562.00
DebitCard - in 5 steps	13 041.00	1 190.00	3 336 711.00	16 229.00	x 1.24	2040.00	x 1.71	1 403 574.00	17 296.00	x 1.33	1 700.00	x 1.43	1 069 516.00
Hanoi - 3 discs	858.00	20.00	68 038.00	1009.00	x 1.18	20.00	x 1.00	26 798.00	935.00	x 1.09	20.00	x 1.00	17 925.00
Hanoi - 5 discs	44 660.00	1 180.00	1 240 553.00	46 162.00	x 1.03	1 160.00	x 0.98	873 147.00	46 656.00	x 1.04	1 190.00	x 1.01	194 719.00
FileSystem - 5 objects	373.00	10.00	52 139.00	479.00	x 1.28	10.00	x 1.00	31 051.00	445.00	x 1.19	10.00	x 1.00	21 839.00
FileSystem - 9 objects	4983.00	110.00	4 708 719.00	5879.00	x 1.18	110.00	x 1.00	4 524 053.00	5738.00	x 1.15	40.00	x 0.36	3 192 895.00
Halmos - 13 people	596.00	40.00	26 572.00	653.00	x 1.10	40.00	x 1.00	5865.00	514.00	x 0.86	40.00	x 1.00	3 193.00
Halmos - 16 people	999.00	20.00	48 031.00	1092.00	x 1.09	10.00	x 0.50	9747.00	856.00	x 0.86	10.00	x 0.50	5401.00

Table 6.5: Comparison of the use of memoization during translation. Reported times are means of 30 runs (preceded by 10 warmup runs).

Problem	With memoization			Without memoization					
	Trans. (in ms)	Solve (in ms)	Mem. used (in GB)	Trans. (in ms)	Solve (in ms)	Mem. used (in GB)			
FileSystem with 5 objects	373.00	10.00	3.85	2035.00	x 5.46	10.00	x 1.00	2.21	x 0.57
FileSystem with 6 objects	690.00	10.00	4.21	4655.00	x 6.75	10.00	x 1.00	2.05	x 0.49
FileSystem with 7 objects	1280.00	20.00	4.40	10 467.00	x 8.18	20.00	x 1.00	2.22	x 0.50
FileSystem with 8 objects	2318.00	30.00	2.80	22 887.00	x 9.87	30.00	x 1.00	2.41	x 0.86
FileSystem with 9 objects	4983.00	110.00	5.12	65 541.00	x 13.15	110.00	x 1.00	2.53	x 0.49

```

1  ...
2  // A dir cannot contain itself
3  forall d: Dir[oid] | no (d[oid as to] & (d[oid as from] |x| ^contents)[to]
4  // All files and dirs are (reflexive-transitive) 'content' of the Root dir
5  (File[oid] + Dir[oid])[oid as to] in (Root[oid as from] |x| *contents)[to]
6  ...

```

Listing 6.3: Excerpt from example given in section 3.2 in chapter 3. During translation memoization is used for the intermediate results.

Memoization To optimize the performance of the translation *memoization* is applied. Memoization is a caching technique that caches the return value of function calls. Whenever a function is called with arguments which it have been give before, the cached result is returned instead of computing the function again.

A requirement for applying memoization is that a function must not have side effects (e.g., writing to a file or database or changing a global variable) since the function is not performed when a cached result can be returned. The translation functions of ALLEALLE meet this requirement since all translations are side-effect free. In many cases applying memoization to the translation functions has a positive effect on the performance of the translation. Memoization does not only cache the end result, it also caches the return values of functions that are called intermediate making the effect even greater.

Listing 6.3 shows an excerpt of our used example from section 3.2 in chapter 3. The constraints encode that directories (`Dir`) cannot contain itself (first constraint, line 3) and that all files and directories must be reachable from the root directory (second constraint, line 5). In the calculation of the first constraint the transitive closure of the content relation is calculated (with the `^content` expression). In the second constraint the reflexive transitive closure of this `content` relation is calculated to constraint that all files and directories including root are reachable from the root directory. When translating the reflexive transitive closure of the content relation (with the `*content` expression) the earlier translated result of the transitive closure can be used since the definition of the reflexive transitive closure translation is the union of the transitive closure of the relation with the identity relation. Hence, using the same constraint multiple times in an specification often does not impact performance much.

The memoization implemented in Rascal is fixed size in terms of memory. Whenever the memoization cache needs to be cleaned (for instance because it reaches it memory limit) the items that are least frequently used are purged, meaning that items used often keep being persistent in the cache. The cache is not persistent between runs: it only resides in memory during a single run of the program.

To measure the effect of memoization we performed an experiment in which we translated and solved the FileSystem specification with an increasing number of tuples in the upper bound of the relations. This experiment was performed using the same setup and configuration as described in the previous section. Table 6.5 shows the outcome of this experiment.

The results clearly show the benefit of applying memoization on translation. The more tuples in the relations upper-bound, the bigger the benefit of memoization. As can be expected, using memoization does have a negative impact on the memory space used, doubling the amount of memory used in most cases. Since Rascal caps the amount of memory for its memoization cache however, the used memory will not grow beyond its limits (which would cause out-of-memory exceptions).

6.3 NEXTEP

NEXTEP (see Chapter 4) is a prototype implementation for automatic repair of run-time state to aid live modeling.** The idea of live modeling is that users get immediate feedback of changes they make to the model without having to restart the execution, provided that there is an interpreter for the model. NEXTEP is our prototype implementation to support this live modeling behavior. NEXTEP allows language engineers to describe their model declaratively (both the meta-model as the run-time behavior) and employs a solver to automatically repair differences in the run-time state before and after a change is made.

NEXTEP uses ALLEALLE as an intermediate language to translate to. Because of this, NEXTEP is also another good case study on the expressiveness of ALLEALLE. NEXTEP is implemented in the Rascal Language Workbench [KvdSV09] and consists out of 991 SLOC. In the rest of this section we will discuss the different parts of the design and implementation of NEXTEP.

6.3.1 Structure of NEXTEP

NEXTEP is designed using the well known *pipe and filters* pattern, a pattern for instance often used by compilers. This means that each step in the pipeline accepts input, performs some operation (e.g., a translation) and hands over its output to the next step in the pipeline.

The general gist of the translation is as follows: A NEXTEP model and a current model instance are translated to an ALLEALLE specification which in turn translates its specification to SMT which is solved using the Z3 SMT solver. Figure 6.4 shows an overview of the different steps that are performed during this translation.

We will discuss each step in more detail using a part of the example introduced in Chapter 4, a DSL for State Machines. Listing 6.4 shows a part from this NEXTEP

**<https://github.com/cwi-swat/live-modeling>

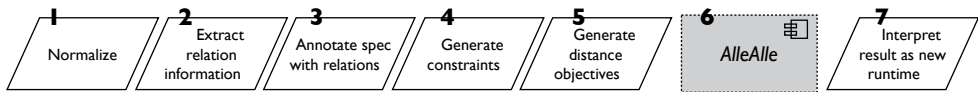


Figure 6.4: Overview of the NEXTEP translation pipeline.

```

1  runtime
2  class Runtime
3    machine: Machine
4    current: State
5    visited: Visit*
6
7  invariants
8    current in machine.states
9    forall s: machine.states | one (visited.state & s)
10   forall v1:visited, v2:visited | v1 != v2 => v1.state != v2.state
11
12  class Visit
13    state: State
14    nr: int
15
16  invariant: nr >= 0
  
```

Listing 6.4: Excerpt of a NEXTEP specifications for live state machine. This excerpt shows the runtime classes declared for the State Machine DSL. Complete specification can be found in Listing 4.2 in Chapter 4

definition containing the specification of the runtime behavior of our State Machine DSL (see Listing 4.2 in Chapter 4 for the complete specification). This example shows the scenario where the user added a new state `locked` to the state machine definition. The addition of this new state requires the runtime instance to be fixed since the meta model of this state machine language (not shown in the listing) dictates that the runtime of each state also contains information on how many times the state has been visited during execution. NEXTEP should thus find a new runtime state where the newly added state `locked` also contains information on how many times it is visited. This information is, as captured in the specification, contained in a `visit` class. Listing 6.5 contains a definition of runtime information of an instance of a State Machine DSL program (i.e., the current state of the evaluation). In Chapter 4 we use the notation x to denote *current runtime* valuation and x' to denote the valuation of the *next runtime*, in other words, the repaired runtime. We will continue to use these notations.

```

1  old runtime
2  Runtime r
3    machine = doors
4    current = closed
5    visited = v1, v2
6
7  Visit v1
8    state = opened
9    nr = 1
10
11 Visit v2
12    state = closed
13    nr = 1

```

Listing 6.5: Excerpt of a NEXTEP current runtime instance (x) of a state machine. Left hand sides ('machine', 'current' and 'visited') are the names of the fields defined in the Runtime class. Right hand sides ('doors', 'closed' and 'v1, v2') are their current values and refer to other instances. States and Machine are defined in the static part and not listed here. The value labels are arbitrary chosen strings

```

1  class Runtime
2    machine: Machine
3    current: State
4    visited: Visit*
5
6    invariants x
7      forall inst : Runtime | inst.current in inst.machine.states
8
9      forall inst : Runtime | forall s:inst.machine.states |
10     one (inst.visited.state & s)
11     forall inst : Runtime | forall v1:inst.visited, v2:inst.visited |
12     v1 != v2 => v1.state != v2.state
13
14  class Visit
15    state: State
16    nr: int
17
18  invariant: forall inst : Visit | inst.nr >= 0

```

Listing 6.6: Result after normalization. Highlighted lines are affected by normalization.

1. Normalize. In the normalization phase constraints in the original NEXTEP model are augmented to aid translation to ALLEALLE later on. For instance, the NEXTEP live state-machine example (Listing 4.2) used in Chapter 4 defines invariants for the class `Runtime`. In the normalization phases these top-level formulas, e.g., `current in machine.states`, are quantified for all instances of the class in which these invariants are defined. The normalized constraint for this example would result in: `forall inst: Runtime | current in inst.machine.states`. This normalization allows for easier relational lookup later on by effectively making each formula ‘operationally complete’. This means that it is not necessary to add extra expressions to a constraint when translating these formula’s to ALLEALLE constraints). Listing 6.6 shows the result of normalization on our running example.

2. Extract relational information. The next step extracts the relational bounds from the normalized model. To extract the bounds both the normalized specification (see Listing 6.6) and the current instance information is used (see Listing 6.5). Section 4.4.2 of Chapter 4 describes the details of this extraction. The outcome of this step is the set of relational definitions (including their bounds) needed for the ALLEALLE specification. Tabel 6.6 shows the relational information extracted from our example specification and runtime instance.

As can be seen in Table 6.6, the bounds of the current valuation relations (denoted by x) have exact bounds, meaning that these relations have exactly those tuples that are defined in their tuple list. The next valuation relations (denoted by x') and the difference relations (denoted by δ) are all upper bound meaning that these relations must have a subset of the tuples defined. Also notice the addition of two new possible visit tuples `v_new_1` and `v_new_2`. Since `Visit` entities are part of the runtime state it could be that a repaired runtime state dictates the addition of a new `Visit` entity (for instance if a new state is added to the model as is the case with our running example). The δ relations encode the differences between the x' and x relations. These δ relations will be used in a later step to minimize the number of difference between the x and x' relations.

3. Annotate spec with relations. This step takes the parsed normalized NEXTEP model and extracted relational information as input. It evaluates the Concrete Syntax Tree (CST) of the normalized model (using recursive tree traversal) and annotates each expression with its resolved relation. This annotation step eases the translation of the NEXTEP constraints to ALLEALLE constraints in the following step. The annotator also acts as a type resolver. Whenever there is a type error in one of the expressions, the annotator will be unable to find the corresponding relation ending up in an error state.

Table 6.6: Extracted relations and their bounds for our running example. x denotes the current valuation, x' denotes the possible new valuation of the relation and δ denotes the difference relation $x' - x$

Relation	Signature	Bound	Tuples
Runtime_ x	rId: id	Exact	<r>
Runtime_ x'	rId: id	Upper	<r>
Runtime_ δ	rId: id	Upper	<r>
Runtime_machine_ x	rId: id , mId: id	Exact	<r, doors>
Runtime_machine_ x'	rId: id , mId: id	Upper	<r, doors>
Runtime_machine_ δ	rId: id , mId: id	Upper	<r, doors>
Runtime_current_ x	rId: id , sId: id	Exact	<r, closed>
Runtime_current_ x'	rId: id , sId: id	Upper	<r, opened>, <r, closed>, <r, locked>
Runtime_current_ δ	rId: id , sId: id	Upper	<r, opened>, <r, closed>, <r, locked>
Runtime_visited_ x	rId: id , vId: id	Exact	<r, v2>, <r, v1>
Runtime_visited_ x'	rId: id , vId: id	Upper	<r, v_new_1>, <r, v_new_2>, <r, v2>, <r, v1>
Runtime_visited_ δ	rId: id , vId: id	Upper	<r, v_new_1>, <r, v_new_2>, <r, v2>, <r, v1>
Visit_ x	vId: id	Exact	<v2>, <v1>
Visit_ x'	vId: id	Upper	<v_new_1>, <v_new_2>, <v2>, <v1>
Visit_ δ	vId: id	Upper	<v_new_1>, <v_new_2>, <v2>, <v1>
Visit_state_ x	vId: id , sId: id	Exact	<v1, opened>, <v2, closed>
Visit_state_ x'	vId: id , sId: id	Upper	<v1, opened>, <v1, closed>, <v1, locked>, <v2, opened>, <v2, closed>, <v2, locked>, <v_new_1, opened>, <v_new_1, closed>, <v_new_1, locked>, <v_new_2, opened>, <v_new_2, closed>, <v_new_2, locked>
Visit_state_ δ	vId: id , sId: id	Upper	<v1, opened>, <v1, closed>, <v1, locked>, <v2, opened>, <v2, closed>, <v2, locked>, <v_new_1, opened>, <v_new_1, closed>, <v_new_1, locked>, <v_new_2, opened>, <v_new_2, closed>, <v_new_2, locked>
Visit_nr_ x	vId: id , val: int	Exact	<v2, 1>, <v1, 1>
Visit_nr_ x'	vId: id , val: int	Upper	<v2, ?>, <v1, ?>, <v_new_1, ?>, <v_new_2, ?>
Visit_nr_ δ	vId: id , val: int	Upper	<v2, ?>, <v1, ?>, <v_new_1, ?>, <v_new_2, ?>

4. Generate constraints. In the next phase the NEXTEP constraints are converted to ALLEALLE constraints. This translation contains two parts: it translates the user defined invariants in the NEXTEP model and it generates relational type constraints. The translation of the user defined invariants is straightforward since the previous steps normalized these invariants and annotated each node in the CST with relational information. Since NEXTEP relies on ALLEALLE for its semantics, the operators in NEXTEP are a subset of those defined in ALLEALLE. Hence, translation of the defined NEXTEP expressions can almost directly be mapped to expressions in ALLEALLE. Listing 6.7 shows an example of the translation of some of the invariants of our running example.

For the relational type constraints the class and field definition in the NEXTEP specification are used. For instance, as mentioned earlier, fields in NEXTEP are

```

1 ...
2 // forall inst : Runtime | inst.current in inst.machine.states
3 forall inst: Runtime_x' |
4   inst |x| Runtime_current_x'[StateId] in
5     (inst |x| Runtime_machine_x'[MachineId]) |x| Machine_states[StateId]
6
7 // forall inst : Runtime | forall s:inst.machine.states | one (inst.visited.state & s)
8 forall inst: Runtime_x' |
9   forall s:(inst |x| Runtime_machine_x'[MachineId]) |x| Machine_states[StateId] |
10    one ((inst |x| Runtime_visited_x'[VisitId]) |x| Visit_state_x'[StateId]) & s
11 ...

```

Listing 6.7: Example of two NEXTEP invariants translated to ALLEALLE constraints.

```

1 ...
2 // class Runtime
3 //   machine: Machine
4 forall r: Runtime_x' | one r |x| Runtime_machine_x'
5 Runtime_machine_x' in (Runtime_x' |x| Machine)
6
7 //   current: State
8 forall r: Runtime_x' | one (r |x| Runtime_currenminimization on these relations is
9   donet_x')
10 Runtime_current_x' in (Runtime_x' |x| State)
11
12 //   visited: Visit*
13 Runtime_visited_x' in (Runtime_x' |x| Visit_x')
14 // Since visited is defined as a set, no extra constraint is needed to
15 // to constraint the size of the relation
16 ...

```

Listing 6.8: Example of the translation of the NEXTEP field and type definition of the Runtime class to ALLEALLE constraints.

translated to binary relations. If a field in the NEXTEP model is defined to hold a single value than we must generate relational constraints restricting it to act as an injective relation. Listing 6.8 shows the translation of the field definitions of the Runtime class. The outcome of this step is a list with ALLEALLE formulas.

5. Generate distance objectives. To complete the translation the distance objectives need to be generated. These objectives encapsulate the heuristic to find a solution for the new run-time state that is closest to the previous run-time state.

In NEXTEP we use the earlier introduced δ -relations to implement the heuristic. We constraint the δ -relations to contain the difference in relational tuples of the current run-time state and the next run-time state. By adding optimization criteria to


```

1 ...
2 Runtime_visited_d =
3   (Runtime_visited_x - Runtime_visited_x') + (Runtime_visited_x' - Runtime_visited_x)
4
5 Runtime_machine_d =
6   (Runtime_machine_x - Runtime_machine_x') + (Runtime_machine_x' - Runtime_machine_x)
7
8 Runtime_current_d =
9   (Runtime_current_x - Runtime_current_x') + (Runtime_current_x' - Runtime_current_x)
10 ...
11 objectives (lex): ..., minimize Runtime_current_d[count()],
12   minimize Runtime_machine_d[count()], minimize Runtime_visited_d[count()], ...

```

Listing 6.9: Example of the constraints added for the δ -relations and the minimization criteria.

minimize the number of tuples in the δ -relations we effectively force the solver to find solutions that are close to the previous run-time state. Since a NEXTEP model can have many fields, the resulting ALLEALLE specification can have many δ -relations and, as a result, also many minimization criteria. These criteria are solved in lexicographical order. Listing 6.9 shows the encoding of the δ -relations and the minimization criteria.

6. Solve using ALLEALLE In this step the completed ALLEALLE specification is send to ALLEALLE which in turn translates the specification to a formula in SMT-LIB which is send to Z3. If the specification can be satisfied, ALLEALLE returns a relational model back to NEXTEP. If it is not satisfiable ALLEALLE returns UNSAT to NEXTEP.

7. Interpret result to construct new run-time state In case a satisfiable model was found NEXTEP maps the found relational tuples back to the level of the NEXTEP model meaning that a NEXTEP model is constructed containing instances of classes and field values. Listing 6.10 shows the output generated by NEXTEP for our running example. As can be seen, ALLEALLE was able to solve this problem and came up with a minimal solution: by adding a new Visit instance, `v_new_1`, which it linked to the user defined, newly added state `locked`.

6.3.2 Live State-Machines: a NEXTEP example.

To test the applicability of NEXTEP we constructed a simple, web-based live-modeling environment for state-machines (438 SLOC). It consists of an interpreter allowing users to interact with their defined state machine and a rudimentary IDE in which users can model the state machine. Figure 6.5 shows a screenshot of this modeling environment.

```

1  new runtime
2    Runtime r
3      visited = v2, v1, v_new_1
4      machine = doors
5      current = closed
6
7    Visit v2
8      nr = 1
9      state = closed
10
11   Visit v1
12     nr = 1
13     state = opened
14
15   Visit v_new_1
16     nr = 0
17     state = locked

```

Listing 6.10: Found NEXTEP output of the live state machine running example

To make use of the automatic run-time state migration there are two requirements for the interpreter:

1. The interpreter must be able to export its current run-time state (variables and values).
2. The interpreter must be able to swap the current run-time state.

The first requirement is needed to be able to construct a NEXTEP model which encapsulates the current state of the run-time model. The second requirement is necessary to replace the complete run-time state with new valuations (as found by NEXTEP). After each change made to the model (or the run-time meta-model) NEXTEP is called to repair the run-time state. If a new model is found, the current run-time state is swapped with this new version.

Performance was never part of this experiment but in general we can remark based on our experience that this method works for small models but it will probably struggle with larger ones adding seconds to the live modeling cycle.

Live Modeling with Nextep demo

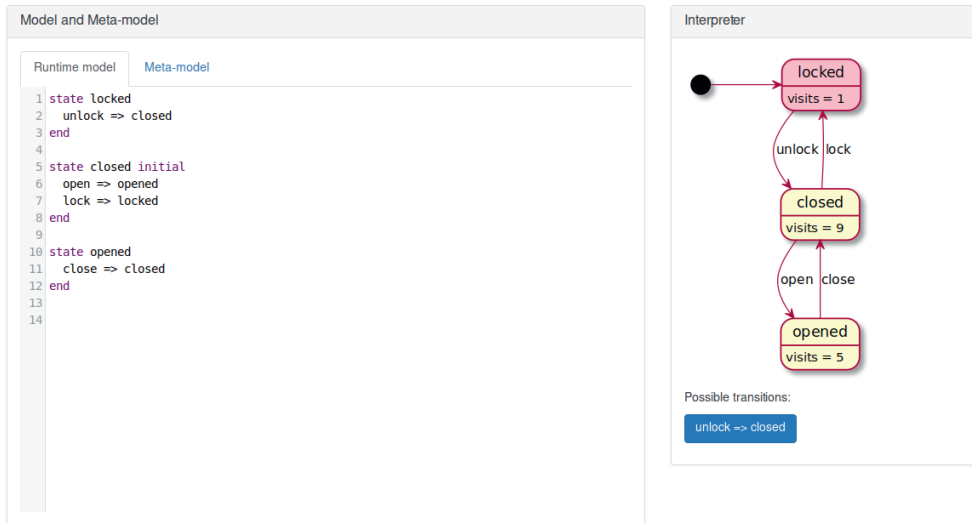


Figure 6.5: Screenshot of the Live State-Machine modeling environment. The red colored state is the current active state. PlantUML was used to visualize the state machine [Pla]. CodeMirror.js was used for the editor [Hav].

6.4 REBEL2

REBEL2 is our second version of a specification language and model checker for a bank.^{††} Like the first version, REBEL2 was based on the formalism of Extended State Machines with data. It allows users to specify problems as separate communicating state machines. This possibility of specifying functionality as separate state machines helps with the decomposition of the problem as a whole. Before we will discuss the overall architectural overview of REBEL2 we will discuss the main differences between the first and the second version of the language.

6.4.1 Comparison to REBEL

In the second version of we took the lessons learned into account as described in section 6.1.1 on the postmortem analysis of REBEL.

Complete symbolic compiler REBEL2 utilizes ALLEALLE as an intermediate language. It translates REBEL2 specifications to ALLEALLE specifications. All

^{††}<https://www.github.com/cwi-swaf/rebel2>

expressions that can be formulated in REBEL2 specifications can be translated to ALLEALLE constraints. Next to encoding the different REBEL2 constructs as ALLEALLE constraints the transition system semantics are also encoded as ALLEALLE constraints. As discussed in Section 5.4.3 of Chapter 5 this translation of the transition system semantics happens similarly as described by Cunha in Alloy [Cun14].

By exploiting ALLEALLE as intermediate language the encoding of REBEL2 specifications as logical constraints is of a higher level of abstraction (e.g., fewer lines of ALLEALLE code required) than our encoding with the first version of REBEL which eased the creation of a symbolic compiler.

Simple and flexible type system REBEL contained many domain specific types that made the encoding financial products easy and natural. The downside was that it made the language rigid for change. New types required changes to the language.

In REBEL2 we simplified this type system. In essence there are only two primitive types, **Integer** and **String**, all other types need to be specified as state machines. The consequence is that fields are either one of the two primitive types or are of the type of other state machines. With special syntax to define an ‘enumeration type’ it is possible to define other well known types such as booleans (see Section 5.4.2 of Chapter 5).

Having no domain specific types also has the consequence that REBEL2 is not a domain specific language for financial products anymore. Instead REBEL2 is more general purpose and allows for the encoding of problems in many different problem domains. The drawback of this choice is that reading REBEL2 specifications can be harder for domain experts since its syntax is less recognizable.

Model checker specification language based on Linear Temporal Logic As discussed in the previous paragraph the encoding of REBEL2 as constraint problem was based on earlier work of encoding transition systems in Alloy [Cun14]. This made implementing a specification language to check properties of interest straightforward. Since LTL is a well known and powerful logic, it allows to express deep properties to check.

6.4.2 Components

REBEL2 has a similar design as the earlier described systems. It contains four different logical components: IDE services, Language, Visualizations and Checker. Figure 6.6 shows these different components in relation to each other. Table 6.7 shows the size in SLOC for each component. We will discuss the different components.

IDE services Like the earlier described software, REBEL2 is written in Rascal [KvdSV09]. Rascal is a *language workbench* [EVV⁺13]. By making use of Rascal’s

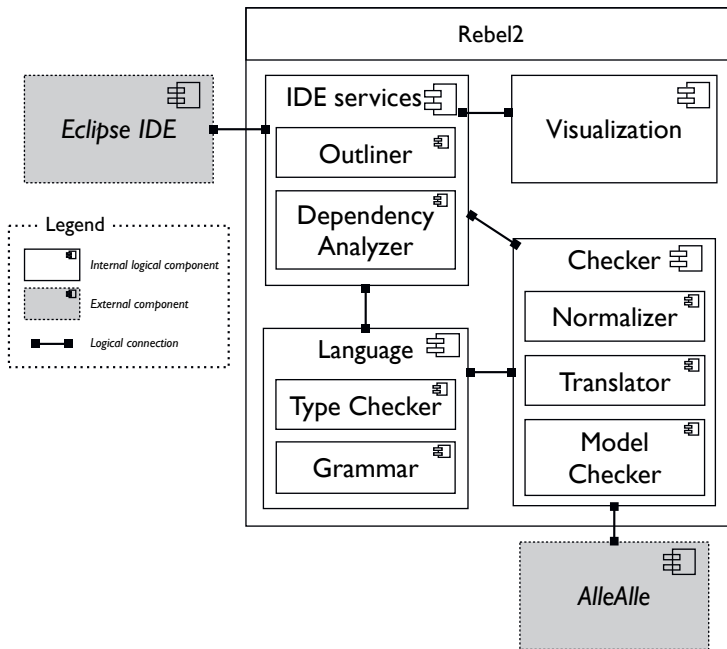


Figure 6.6: Logical components of REBEL2

Table 6.7: Overview of Source Lines Of Code (SLOC) for the different logical components of REBEL2. Source lines of tests are not counted.

Component	SLOC
IDE Services	220
Visualizations	404
Language	1295
Model Checker	2469
Total	4388

support for IDE integration we constructed an IDE for REBEL2. Rascal eases integration into Eclipse by offering high-level abstractions for IDE integration (such as hyperlinking, outlines, error messaging, hover documentation, etc.) and having out-of-the-box techniques for specifying grammars of which parser are automatically generated. Using these techniques we implemented IDE features for REBEL2 such as syntax highlighting, outline, hyperlinking by jumping from use to definition, error

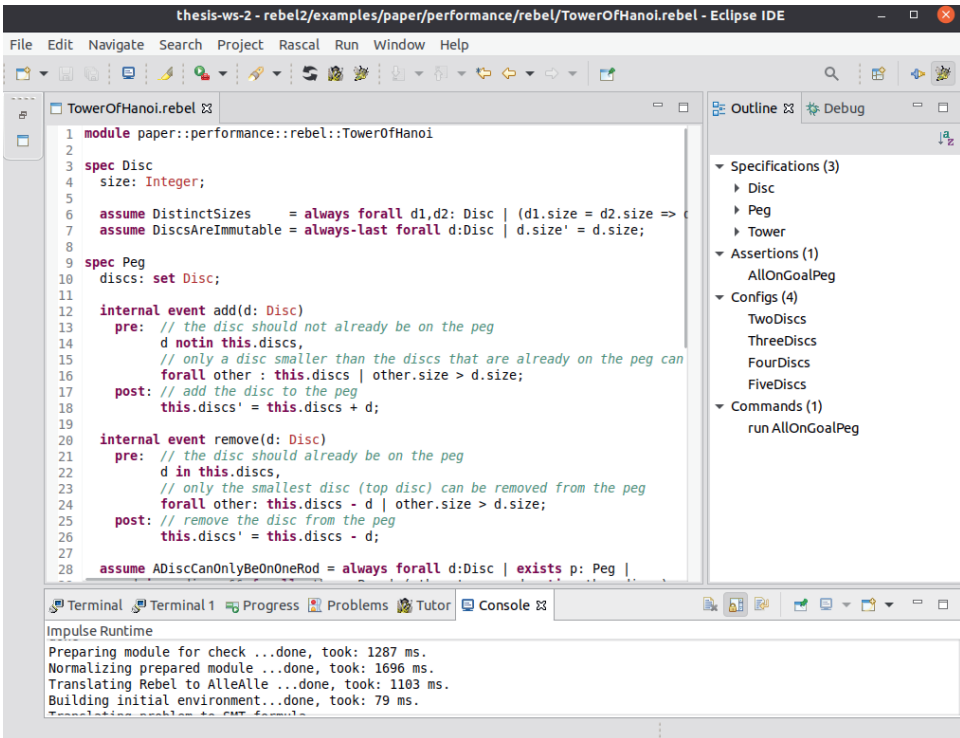


Figure 6.7: Example of the REBEL2 IDE displaying the Tower of Hanoi specification.

messaging and warning. Most operations that can be done by users (visualizing a specification, checking an assumption, etc.) are done via this component. It coordinates the interaction between the underlying Language, Checker and Visualization components. Figure 6.7 shows a screenshot of the REBEL2 IDE.

Language The *language* components contains the grammar definition and type checker. The type checker is build using *TypePal* [Kli15]. *TypePal* is a Rascal framework with which a declarative type checker can be implemented on top of a grammar. The result of type checking via this framework is a record, called a *TypeModel* in *TypePal*, containing type information and possible error messages. This *TypeModel* is used by many of the other (sub)components. For instance, when translating REBEL2 specifications to ALLEALLE problems. The size of the implemented type checker is 942 SLOC.

Model Checker The model checker component is responsible of translating REBEL2 specifications to an ALLEALLE specification. To perform this translation it takes the

Found trace for check `OptingInForDailyTestingEndsIsolation`

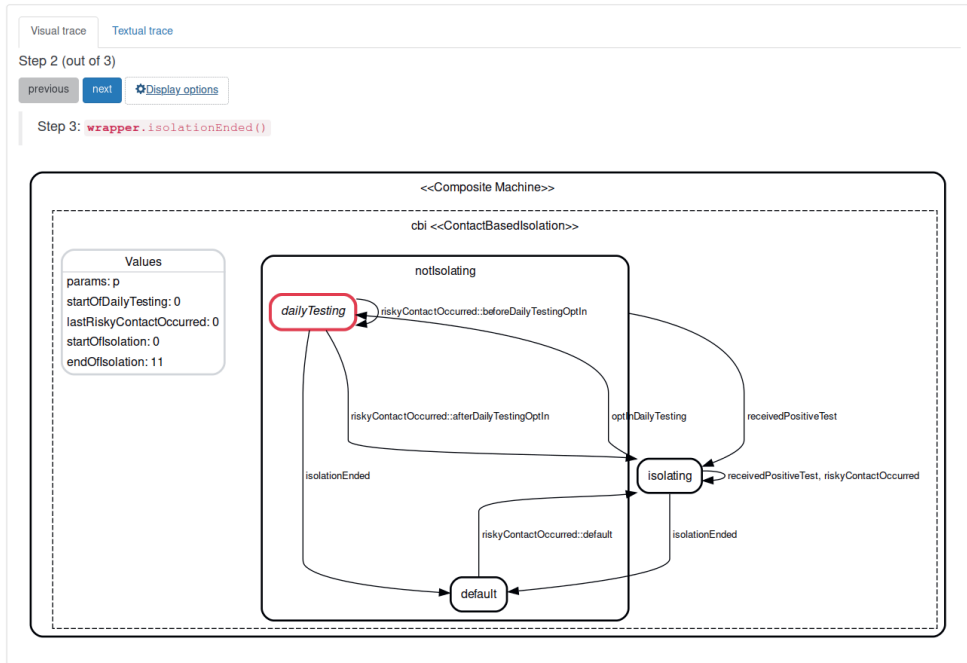


Figure 6.8: Visualizing a found trace in a REBEL2 formalization of UK’s COVID-19 isolation rules which were in place during the COVID-epidemic (2020-2022).

REBEL2 specifications, the type model and the assumption of interest as input. The outcome can either be that the assumption can not be verified or that a trace has been found. The model checker returns this trace as an internal data structure (i.e., as an ADT). Section 5.4 of chapter 5 describes the different steps of the translation from REBEL2 to ALLEALLE.

Interactive visualization The interactive, web-based visualization allows the user to step through the found trace. Figure 6.8 shows an example of the visualizer. The Javascript library ‘State Machine cat’ is used to render the state machines [Ver15].

In each step the current state in the state machines is updated together with the assigned values. Navigating through the trace is done using the previous and next buttons. Next to the state and data, the event which is raised between the steps is also shown. Hovering over the ‘next’-button highlights the state machines will take part in the next transition.

6.5 CONCLUSION

During the course of our research we have created four different DSL's. Two of these DSL's (REBEL and REBEL2) were created to support the use of lightweight formal methods within the context of an enterprise. One DSL, ALLEALLE, was created to act as an intermediate language to ease the transition from other DSL's to SMT solvers. The last DSL, NEXTEP, was amongst other reasons created to validate the expressiveness of ALLEALLE.

REBEL is our first iteration of a DSL to define banking products. It allows for the checking of some user defined properties. The problem with REBEL is that it has an incomplete mapping to SMT and it has a too restrictive type system.

To overcome some of the problems we faced during the development of REBEL we developed ALLEALLE. ALLEALLE is created to act as an intermediate language between DSL's such as REBEL and SMT. ALLEALLE is based on Codd's relational algebra and allows for the definition of relational problems. The translation of ALLEALLE is complete meaning that all constructs of ALLEALLE can be automatically translated and expressed in SMT.

To test ALLEALLE (amongst other things) NEXTEP was created. NEXTEP is a formalism that allows for the definition of the static and dynamic semantics of a DSL. This definition, together with run-time state information, can be utilized during execution to automatically repair the run-time state when the program changes.

Lastly, we created REBEL2, a new iteration of REBEL. Unlike REBEL, REBEL2 has a complete translation to ALLEALLE meaning that all properties that can be expressed in REBEL2 can be (model) checked by ALLEALLE. Next to that, REBEL2 implements a novel idea that lets users apply mocking to their specifications enabling model checking on parts of the full system specification.

All our components were created using the language workbench Rascal [KvdSV09]. Using such a language workbench has been instrumental in the creation of our DSL's. It enabled us to quickly prototype certain ideas without having to spend time on creating parsers and basic IDE support. The prototypes we created could be used to validate with our collaborating partner whether a DSL was understandable and expressive enough. Having a quick turn-around time (from idea to DSL) was very beneficial.

6.5.1 *Future engineering work*

All prototypes lack run time speed in their current form. This could hamper adoption by a broader audience. There are, however still many performance improvements possible but these would require a considerable engineering effort.

Another direction for future work that could help adoption would be to support different IDE's. The current implementations of ALLEALLE and REBEL2 are tied to

the Eclipse IDE.[‡] This IDE has seen declining user numbers over the last couple of years. One way forward would be to utilize the Language Server Protocol (LSP).[§] LSP allows for the decoupling of language IDE services (such as jump to definition and auto completion) from editor integration making it easier to integrate a language in multiple editors. Fortunately Rascal itself moved toward the use of LSP. Offering language service for REBEL2 and ALLEALLE to other editors should therefore be straightforward.

[‡]<https://www.eclipse.org/ide/>

[§]<https://microsoft.github.io/language-server-protocol/>

CONCLUSION

Enterprise Software Systems are complex systems consisting out of many different applications, written in different languages, over different eras, by different people. Evolving such a system is a daunting task. One of the problems encountered during this task is that the intended working of the system as a whole is often not captured in a precise description. If this knowledge *is* captured it is often informal, incomplete and outdated or became tacit knowledge to the people working in these organizations.

Making this knowledge explicit and precise in the form of formal specifications can help in understanding and verifying its intended working. Such a specification can be used to formally verify the correct behavior of a system a priori instead of testing for correctness a posteriori. The problem however with the application of these formal specifications is that the cost of fully formalizing a system is considered too high by many professionals in the field [GM20].

In the late nineties Jackson et al. proposed the creation and use of so called *lightweight formal methods*. They argued that instead of focusing on full formalization the emphasis should be on partiality: partiality in (specification) language, partiality in modeling, partiality in analysis and partiality in composition. These different partialities offer practitioners tools to make different trade-offs depending on the problem at hand.

Our work is done in collaboration with the ING bank, a large, Dutch national bank. The ING experiences similar challenges as described above. Their IT landscape is vast and wide containing many applications which evolved over a time period of sixty years. Fully formalizing such a system is bordering the impossible.

To this end we explored the use of lightweight formal methods in the context of the ING bank leading to our general research question:

General Research Question

What is the impact of different choices along the axes of partiality on the design and verification of enterprise software systems using lightweight formal methods?

To gain insight into this general question we formulated questions placed amongst the different axes of partiality. In the remainder of this chapter we will revisit these questions and consolidate the conclusions from the different chapters.

Research Question 1

How can we design specification languages such that they are expressive enough to specify problems in the enterprise domain while still able to perform automatic analysis?

In chapter 2 we described the design and (prototype) implementation of the REBEL language and its Interactive Specification Environment (ISE). This version of REBEL was a domain specific language and created for the definition of financial products [MHS05]. This was especially reflected in its domain specific types (such as **Money**, **IBAN** and **Term**). We observed (qualitatively) that having domain specific types helped non-technical readers to better understand these specifications.* The downside of this approach was that implementing automatic reasoning for these domain specific types was complex (see chapter 6). Our analysis of this problem is that this first iteration of the language was not designed with full automated reasoning in mind from the start. Adding an efficient encoding for the different domain specific types as an afterthought has proven to be a big hurdle (as is also remarked by Jackson et al. [JW96]).

In chapter 5 we described a second iteration of the REBEL language named REBEL2. This new version was more generic by nature but was created with automatic reasoning in mind from the early stages. The types that were domain specific in the first version, were now created as specifications in the language itself. The benefit of this approach was that the constructs that needed to be translated to the automatic reasoning engine were less than for the first version of the language. Experimentation with the expressiveness of this second version of the language showed that it was possible to specify similar banking products as specified in the first version of the language while being able to check for user defined properties in (parts of) the specifications. The downside of this approach being that the specifications written in REBEL2 are further away from the problem domain which could hinder understanding for domain experts in the field.

*Research by van Gasteren showed that natural language documents derived from REBEL specification were considered even better understandable by bank employees than the specifications themselves [vGas16].

7.2 RESEARCH QUESTION 2: PARTIALITY OF MODELING

Research Question 2

How can we manipulate the cost of modeling for different parts of an enterprise system while preserving the positive impact of specifying?

This question was fueled by the notion that there is value in specifying even if these specifications are not used for verification (see chapter 1, section 1.7.2).

In chapter 5 we explored the concepts of mocking in the context of formal specifications. By enabling users to either remove parts of a specification (using the **forget** keyword) or replacing parts of the specification with different behavior (using the **mock** keyword) it became possible to perform model checking on parts of the specification of the system without altering the original specifications. We argue that this paves the way for more hybrid specification approaches where a user can choose which parts of a system design are important enough to justify the cost of full formalization and verification while other parts can be merely specified.

The potential downside of this method is that this method does not give the user guarantees that the chosen specification replacements (or removals) are valid abstractions of the problem. It could be that the user introduces behavior in a mocked specification that is not present in the original specification. This can lead to false positives while model checking, meaning that a user can get the impression that a specification contains the desired behavior while in practice it does not. This judgment call is left to the user. Still we argue that this method has benefits since it gives the user a method to balance the trade-off of full versus partial formalization. Next to that, this false sense of correctness is also present when performing traditional unit testing using mocks [MFCoo]. This technique, however, is widely used and recognized as valuable [SAB⁺17].

7.3 RESEARCH QUESTION 3: PARTIALITY OF ANALYSIS

Research Question 3

How can we extend the current state of the art in relational model finding in such a way that it is possible to efficiently reason about other theories such as integer arithmetic?

Relational model finding is a successful technique for solving problems that are of a structural nature as often found in software design. KODKOD is a very efficient implementation of this techniques [TJ07b]. The state of the art utilizes SAT solvers as a solver back-end. While SAT solvers are very efficient when it comes to solving

boolean problems, they are less efficient in solving problems that require reasoning over other domains such as integer arithmetic.

In chapter 3 we introduced ALLEALLE. ALLEALLE is based on Codd's relational algebra [Cod70]. Utilizing Codd's algebra allowed for the use of multi sorted relations while preserving the well known semantics of relational operators. This allowed for the encoding of problems that were both relational and numerical by nature. An algorithm was introduced to translate this hybrid relational and numerical problem to an SMT formula which in turn could be solved by an SMT solver, Z3 [DBo8]. The benefit of this approach is efficient reasoning on these hybrid problems. ALLEALLE out performed KODKOD on these type of problems. KODKOD however was more efficient in reasoning on pure relational problems.

7.4 RESEARCH QUESTION 4: PARTIALITY OF COMPOSITION

Research Question 4

How can we lift the problem of composition to the language level such that the user is able to specify different compositions during specification and analysis?

As advocated by Jackson et al. specifying a complex system will most likely result in many partial specifications describing different parts. Although it is possible to perform automatic reasoning on the different parts, it is hard to automatically reason about different cross sections of these specifications. This would require new specifications describing the cross section of interest.

In chapter 5 we explore this problem and introduce a mechanism to perform impromptu compositions of specifications. This is achieved with the use of two language concepts:

1. By allowing different configurations (bounds) to be used for checking user defined properties of interest.
2. With the earlier mentioned constructs of **forget** and **mock** which allow for the removal or replacement of a specification for the purpose checking a user defined property.

The combination of these two concepts allow for ad hoc compositions of specifications when performing model checking, giving more flexibility to the user to verify properties on different constellations of specifications.

7.5 FUTURE DIRECTIONS

During our work we identified many different potential directions for future work of which we will highlight the most important here:

Solver aided design for specification languages Designing and construction languages and symbolic compilers for specification languages such that automatic reasoning can be performed is a difficult problem. It poses a significant engineering challenge and a large time investment. One possible way to lessen this burden is by making use of a framework that helps creating *solver aided languages*. Rosette is an example of such a framework [TB13]. By using such a framework for the definition of a specification language such as described in this work some automated reasoning techniques could be achieved with low cost. For instance, the power of Rosette could be used to perform some sort of equivalence checking whether introduced mocks display similar behavior than their full counterparts (see chapter 5).

Multi-aspect modeling In our work we focused on the composition of different functional specification of a system. Besides this we can imagine that for some parts of a system it would be of interest to specify non-functional properties such as performance or security aspects as well. How to compose these different views of the system is an open question.

Another angle is to investigate methods that would allow for the composition of different verification techniques in a single set of specifications. For instance, for some parts of a system performing a shallow (and often cheap) verification technique such as static checking might suffice while other parts that are considered critical require the formal proof of some safety properties.

Improving relational model finding with SMT solvers There are several directions to explore that could increase the effectiveness of ALLEALLE. Firstly, the current implementation of ALLEALLE lacks symmetry breaking [TJ07b]. Symmetry breaking is a technique that prevents searching for symmetric solutions of a problem. Although it is known how to perform this for the current SAT based solution (using an algorithm known as greedy base partitioning and the addition of symmetry breaking predicates) it is an open question how this should be implemented for multi sorted relations.

Secondly, the current implementation does not offer the user any guidance on the reason why a specification is satisfiable or not. It is left to the user to find out why a specified property is not satisfiable. There are known ways to provide (some) feedback to the user in these situations. For instance, Torlak describes the use of *unsatisfiability core extraction*, a method to derive the minimal set of clauses that make the formula unsatisfiable. This information is then lifted to the level of the relational specification and shown to the user. How this is to be done for the relational logic of ALLEALLE is an open question.

Another way to supply information on reasoning is to provide *provenance* for the existence of certain tuples in a relation [NDD⁺17]. This would allow users to interactively query the specification to automatically deduce why certain tuples are

or are not allowed. Again, how to do this in the light of ALLEALLE's multi sorted relations is to be seen.

7.6 ADVICE FOR OUR COLLABORATING PARTNER

Our research was inspired by the challenges faced by our industrial partner. In the final paragraphs of this thesis we would like to offer some further advice to our collaborating partner, the ING bank, we derived from our research and experiences.

Creating specifications has value During this work we closely worked together with different employees of the bank. What became clear time after time again is that the process of specifying holds great value. By systematic elicitation and the need to concisely notate this gained knowledge in a specification many unknowns, unclarities and assumptions already arose. To this end we advise that the process of writing (formal) specifications to describe the functional behavior should become part of the development cycle. Whether this should be done for existing applications depends on how critical it is versus how often it changes. Whenever a large refactoring is to be performed, it could be wise to create specifications up front such that it can be used to check for conformance later.

Use specifications for testing purposes The created specifications can be used for testing purposes to check whether applications conform to their specifications. This could be done in an automated fashion using techniques such as model based testing [DJK⁺99]. But even if no automated techniques are used, using the specifications to guide the testing (unit, integration or manual system tests) hold value and can help in spotting inconsistencies early in the process.

Continue the path of application generation from specifications The ultimate goal of writing formal specifications of the system was to generate new software that is correct-by-construction. In a parallel track during our research great strides were made in this area. Soethout et al. researched sound methods for generating systems from REBEL specifications [SSV20; SvdSV21a]. These generated systems are highly distributed and scalable by nature while staying true to the semantics of REBEL. To the author of this thesis, this seems as a promising method to create the correct and scalable applications of the future!

BIBLIOGRAPHY

- [AJM⁺00] M. D. Aagaard, R. B. Jones, T. F. Melham, J. W. O’leary, and C.-J. H. Seger. “A methodology for large-scale hardware verification”. In: *International Conference on Formal Methods in Computer-Aided Design*. Springer. 2000, pp. 300–319 (cit. on p. 3).
- [ATC⁺11] P. Abate, R. Treinen, R. D. Cosmo, and S. Zacchiroli. “MPM : a modular package manager”. In: *CBSE*. ACM, 2011, pp. 179–188 (cit. on pp. 32, 53, 57).
- [Abr96] J. Abrial. *The B-Book: Assigning programs to meaning*. Cambridge University Press, 1996 (cit. on pp. 2, 27, 28).
- [AHO7] J.-R. Abrial and S. Hallerstede. “Refinement, decomposition, and instantiation of discrete models: Application to Event-B”. In: *Fundamenta Informaticae* 77.1-2 (2007), pp. 1–28 (cit. on pp. 91, 114).
- [AL98] S. Agerholm and P.G. Larsen. “A lightweight approach to formal methods”. In: *International Workshop on Current Trends in Applied Formal Methods*. Springer. 1998, pp. 168–183 (cit. on pp. 5, 6).
- [AU79] A. Aho and J. Ullman. “Universality of data retrieval languages”. In: *POPL*. ACM. 1979, pp. 110–119 (cit. on p. 36).
- [ABG⁺20] P. Arcaini, S. Bonfanti, A. Gargantini, E. Riccobene, and P. Scandurra. “Modelling an Automotive Software-Intensive System with Adaptive Features Using ASMETA”. In: *International Conference on Rigorous State-Based Methods*. Springer. 2020, pp. 302–317 (cit. on pp. 109, 110).
- [AvDR95] B. Arnold, A. van Deursen, and M. Res. “An algebraic specification of a language for describing financial products”. In: *ICSE-17 Workshop on Formal Methods Application in Software Engineering*. 1995, pp. 6–13 (cit. on p. 28).
- [BPH⁺13] B. Bajić-Bizumić, C. Petitpierre, H. Huynh, and A. Wegmann. “A model-driven environment for service design, simulation and prototyping”. In: *ESS*. Springer. 2013, pp. 200–214 (cit. on p. 31).
- [BRB⁺16] K. Bansal, A. Reynolds, C. Barrett, and C. Tinelli. “A new decision procedure for finite sets and cardinality constraints in SMT”. In: *CAV*. Springer. 2016, pp. 82–98 (cit. on p. 61).
- [BCD⁺11] C. Barrett, C. Conway, M. Deters, L. Hadarean, J. D., T. King, A. Reynolds, and C. Tinelli. “CVC4”. In: *CAV*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, July 2011, pp. 171–177 (cit. on pp. 58, 61).

- [BST10] C. Barrett, A. Stump, and C. Tinelli. *The smt-lib standard: Version 2.0*. Tech. rep. Department of Computer Science, The University of Iowa, 2010 (cit. on p. 42).
- [BBF⁺99] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. “METEOR: A successful application of B in a large project”. In: *International Symposium on Formal Methods*. Springer. 1999, pp. 369–387 (cit. on p. 2).
- [BBH⁺74] H. Bekić, D. Bjørner, W. Henhapl, C. Jones, and P. Lucas. *A formal definition of a PL/i subset*. TR (IBM Laboratory Vienna) dl. 2. IBM Laboratory Vienna, 1974 (cit. on p. 5).
- [Big89] T. J. Biggerstaff. “Design recovery for maintenance and reuse”. In: *Computer* 22.7 (1989), pp. 36–49 (cit. on p. 2).
- [BGW16] R. Bill, M. Gogolla, and M. Wimmer. “On Leveraging UML/OCL for Model Synchronization”. In: *Proceedings of the 10th Workshop on Models and Evolution co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016*. Ed. by T. Mayerhofer, A. Pierantonio, B. Schätz, and D. Tamzalit. Vol. 1706. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 20–29 (cit. on p. 85).
- [BJ] D. Bjørner and C. B. Jones. *The Vienna development method: The meta-language*. Springer (cit. on pp. 2, 5).
- [BPF15] N. Bjørner, A. Phan, and L. Fleckenstein. “vZ - An Optimizing SMT Solver”. In: *TACAS*. Vol. 15. 2015, pp. 194–199 (cit. on pp. 32, 39, 42, 59).
- [BTV09] N. Bjørner, N. Tillmann, and A. Voronkov. “Path feasibility analysis for string-manipulating programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*. Springer. 2009, pp. 307–321 (cit. on p. 123).
- [Bör98] E. Börger. “High level system design and analysis using abstract state machines”. In: *International Workshop on Current Trends in Applied Formal Methods*. Springer. 1998, pp. 1–43 (cit. on p. 114).
- [BJA⁺21] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, et al. “Using lightweight formal methods to validate a key-value storage node in Amazon S3”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 836–850 (cit. on p. 6).

- [BLC⁺18] E. Bousse, D. Leroy, B. Combemale, M. Wimmer, and B. Baudry. “Omniscient debugging for executable DSLs”. In: *Journal of Systems and Software* 137 (2018), pp. 261–288. DOI: 10.1016/j.jss.2017.11.025 (cit. on p. 86).
- [BMC⁺17] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry. “Advanced and efficient execution trace management for executable domain-specific modeling languages”. In: *Software & Systems Modeling* (2017), pp. 1–37 (cit. on p. 86).
- [BCR⁺14] J. Brunel, D. Chemouil, L. Rioux, M. Bakkali, and F. Vallée. “A viewpoint-based approach for formal safety & security assessment of system architectures”. In: *MoDeVva*. Vol. 1235. 2014, pp. 39–48 (cit. on p. 31).
- [BCC⁺18] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo. “The electrom analyzer: model checking relational first-order temporal specifications”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 884–887 (cit. on p. 5).
- [BFdH⁺13] S. Burckhardt, M. Fähndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. “It’s alive! continuous feedback in UI programming”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 95–104. DOI: 10.1145/2462156.2462170 (cit. on p. 65).
- [CCR14] J. Cabot, R. Clarisó, and D. Riera. “On the verification of UML/OCL class diagrams using constraint programming”. In: *Journal of Systems and Software* 93 (2014), pp. 1–23. DOI: 10.1016/j.jss.2014.03.023 (cit. on p. 86).
- [CCO⁺04] S. Chaki, E.M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. “State/event-based software model checking”. In: *International Conference on Integrated Formal Methods*. Springer. 2004, pp. 128–147 (cit. on pp. 99–101).
- [Cha05] R.N. Charette. “Why software fails”. In: *IEEE spectrum* 42.9 (2005), pp. 42–49 (cit. on pp. 1, 4).
- [CFS] M. Christerson, D. Frankel, and T. Schiller. *Financial Domain-Specific Language Listing*. <http://www.dslfin.org/resources.html>. Accessed: 4-8-2016 (cit. on p. 28).
- [CGS11] A. Cimatti, A. Griggio, and R. Sebastiani. “Computing small unsatisfiable cores in satisfiability modulo theories”. In: *Journal of Artificial Intelligence Research* 40 (2011), pp. 701–718 (cit. on p. 26).

- [CS03] K. Claessen and N. Sörensson. “New techniques that improve MACE-style finite model finding”. In: *CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*. Citeseer. 2003, pp. 11–27 (cit. on pp. 60, 62).
- [CH00] K. Claessen and J. Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs”. In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 2000, pp. 268–279 (cit. on p. 6).
- [CHV⁺18] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. *Handbook of model checking*. Vol. 10. Springer, 2018 (cit. on pp. 3, 91).
- [CES09] E. Clarke, E. Emerson, and J. Sifakis. “Model Checking : Algorithmic Verification and Debugging”. In: *Communications of the ACM* 52.11 (2009), pp. 74–84 (cit. on p. 23).
- [CGK⁺99] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT press, 1999 (cit. on p. 91).
- [Cod70] E. Codd. “A relational model of data for large shared data banks”. In: *Communications of the ACM* 13.6 (1970), pp. 377–387 (cit. on pp. 31, 38, 39, 150).
- [Cod83] E. F. Codd. “A relational model of data for large shared data banks”. In: *Communications of the ACM* 26.1 (1983), pp. 64–69 (cit. on p. 107).
- [CCP12] B. Combemale, X. Crégut, and M. Pantel. “A design pattern to build executable DSMLs and associated V&V tools”. In: *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*. Vol. 1. IEEE. 2012, pp. 282–287 (cit. on p. 66).
- [Cos05] R. D. Cosmo. *EDOS deliverable WP2-D2 . 1 : Report on Formal Management of Software Dependencies*. Tech. rep. 2005. URL: <https://hal.inria.fr/hal-00697463/document> (cit. on p. 57).
- [CD08] M. L. Crane and J. Dingel. “Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities”. In: *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*. 2008, p. 8. DOI: 10.1145/1463788.1463799 (cit. on p. 86).
- [Cun14] A. Cunha. “Bounded model checking of temporal formulas with Alloy”. In: *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer. 2014, pp. 303–308 (cit. on pp. 107, 140).

- [CML20] A. Cunha, N. Macedo, and C. Liu. “Validating Multiple Variants of an Automotive Light System with Electrum”. In: *International Conference on Rigorous State-Based Methods*. Springer. 2020, pp. 318–334 (cit. on pp. 109, 110).
- [DJK⁺99] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton, and B. Horowitz. “Model-based testing in practice”. In: *Proceedings of the 1999 International Conference on Software Engineering 1999*. May (1999), pp. 285–294 (cit. on pp. 26, 152).
- [Dat94] C. Date. *An Introduction to Database Systems*. 6th. Reading, MA, Addison-Wesley, 1994, p. 839 (cit. on pp. 38, 107).
- [DGW⁺14] J. Davies, J. Gibbons, J. Welch, and E. Crichton. “Model-driven engineering of information systems: 10 years and 1000 versions”. In: *Science of Computer Programming* 89 (Sept. 2014), pp. 88–104 (cit. on p. 28).
- [DBo8] L. De Moura and N. Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340 (cit. on pp. 16, 32, 58, 66, 73, 92, 102, 150).
- [DP13] L. De Moura and G. O. Passmore. “The strategy challenge in SMT solving”. In: *Automated Reasoning and Mathematics*. Springer, 2013, pp. 15–44 (cit. on p. 127).
- [DS95] G. Dedene and M. Snoeck. “Formal deadlock elimination in an object oriented conceptual schema”. In: *Data & Knowledge Engineering* 15.1 (1995), pp. 1–30 (cit. on p. 28).
- [DSD92] G. Dedene, M. Snoeck, and A. Depuydt. “On generalisation/specialisation hierarchies in MERODE-object-oriented business modeling”. In: *DTEW Research Report 9219* (1992) (cit. on p. 6).
- [Den09] G. D. Dennis. “A relational framework for bounded program verification”. PhD thesis. Massachusetts Institute of Technology, 2009 (cit. on p. 92).
- [DKo7] D. Dotan and A. Kirshin. “Debugging and testing behavioral UML models”. In: *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. 2007, pp. 838–839. DOI: 10.1145/1297846.1297915 (cit. on p. 86).
- [DKS⁺00] G. Droschl, W. Kuhn, G. Sonneck, and M. Thuswald. “A formal methods case study: Using light-weight VDM for the development of a security system module”. In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2000, pp. 187–197 (cit. on pp. 6, 8).

- [EHP⁺18] R. Eilers, J. Hage, W. Prasetya, and J. Bosman. “Fine-Grained Model Slicing for Rebel”. In: *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 235–244 (cit. on p. 114).
- [Ekeo2] S. Eker. “Single elementary associative-commutative matching”. In: *Journal of Automated Reasoning* 28.1 (2002), pp. 35–51 (cit. on p. 126).
- [ET11] A. El Ghazi and M. Taghdiri. “Relational reasoning via SMT solving”. In: *FM*. Springer, 2011, pp. 133–148 (cit. on p. 61).
- [EGK⁺18] F. Erata, A. Goknil, I. Kurtev, and B. Tekinerdogan. “AlloyInEcore: Embedding of First-Order Relational Logic into Meta-Object Facility for Automated Model Reasoning”. In: *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 18)*. 2018. DOI: 10.1145/3236024.3264588 (cit. on p. 86).
- [EVV⁺13] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. “The state of the art in language workbenches”. In: *International Conference on Software Language Engineering*. Springer, 2013, pp. 197–217 (cit. on p. 140).
- [FGM⁺07] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. “Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem”. In: *ACM Trans. Program. Lang. Syst.* 29.3 (2007), p. 17. DOI: 10.1145/1232420.1232424 (cit. on p. 85).
- [Fow02] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002 (cit. on p. 1).
- [FGL⁺05] M. F. Frias, J. P. Galeotti, C. G. López Pombo, and N. M. Aguirre. “DynAlloy: upgrading alloy with actions”. In: *Proceedings of the 27th international conference on Software engineering*. 2005, pp. 442–451 (cit. on p. 113).
- [GRSo4] A. Gimblett, M. Roggenbach, and H. Schlingloff. “Towards a formal specification of electronic payment systems in CSP-CASL”. In: *Recent Trends in Algebraic Development Techniques*. Springer, 2004, pp. 61–78 (cit. on pp. 7, 29).
- [GM20] M. Gleirscher and D. Marmsoler. “Formal methods in dependable systems engineering: a survey of professionals from Europe and North America”. In: *Empirical Software Engineering* 25.6 (2020), pp. 4473–4546 (cit. on pp. 3, 147).
- [GM14] J. F. Groote and M. R. Mousavi. *Modeling and analysis of communicating systems*. MIT press, 2014 (cit. on pp. 2, 91, 113).

- [GGL⁺14] D. Grunwald, C. Gladisch, T. Liu, M. Taghdiri, and S. Tyszberowicz. “Generating JML specifications from alloy expressions”. In: *Hardware and Software: Verification and Testing: 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings* 10. Springer. 2014, pp. 99–115 (cit. on p. 31).
- [GB95] Y. Gurevich and E. Börger. “Evolving algebras 1993: Lipari guide”. In: *Evolving Algebras* 40 (1995) (cit. on p. 2).
- [Hav] M. Haverbeke. CodeMirror. <https://codemirror.net/>. Accessed: 2022-01-09 (cit. on p. 139).
- [HL91] G. J. Holzmann and W. S. Lieberman. *Design and validation of computer protocols*. Vol. 512. Prentice hall Englewood Cliffs, 1991 (cit. on p. 2).
- [HA00] J. Horl and B. K. Aichernig. “Validating voice communication requirements using lightweight formal methods”. In: *IEEE Software* 17.3 (2000), pp. 21–27 (cit. on p. 7).
- [HR20] F. Houdek and A. Raschke. “Adaptive Exterior Light and Speed Control System”. In: *International Conference on Rigorous State-Based Methods*. Springer. 2020, pp. 281–301 (cit. on p. 108).
- [Jac01] D. Jackson. “Lightweight formal methods”. In: *FME 2001: Formal Methods for Increasing Software Productivity*. Springer, 2001, pp. 1–1 (cit. on pp. 16, 27).
- [Jac02a] D. Jackson. “Alloy: a lightweight object modelling notation”. In: *ACM Transactions on Software Engineering and Methodology* 11.2 (2002), pp. 256–290 (cit. on pp. 5, 27).
- [Jac12] D. Jackson. *Software Abstractions - Logic, Language, and Analysis*. Revised. MIT press, 2012, p. 336 (cit. on pp. 5, 23, 33, 60, 73, 102, 113, 119).
- [JSS00] D. Jackson, I. Schechter, and H. Shlyachter. “Alcoa: the alloy constraint analyzer”. In: *ICSE*. ACM. 2000, pp. 730–733 (cit. on p. 5).
- [Jac02b] D. Jackson. “Alloy: a lightweight object modelling notation”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.2 (2002), pp. 256–290 (cit. on pp. 36, 92, 113, 119).
- [JW96] D. Jackson and J. Wing. “Lightweight formal methods”. In: *IEEE Comput.* 29.4 (1996), pp. 21–22 (cit. on pp. 3, 7–9, 92, 147, 148, 150).
- [KGN⁺09] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, et al. “Replacing Testing with Formal Verification in Intel Core™ i7 Processor Execution Engine Validation”. In: *International Conference on Computer Aided Verification*. Springer. 2009, pp. 414–429 (cit. on p. 3).

- [KJ09] E. Kang and D. Jackson. “Designing and analyzing a flash file system with Alloy.” In: *Int. J. Softw. Informatics* 3.2-3 (2009), pp. 129–148 (cit. on p. 5).
- [KN86] D. Kapur and P. Narendran. “NP-completeness of the set unification and matching problems”. In: *International conference on automated deduction*. Springer, 1986, pp. 489–495 (cit. on p. 126).
- [Kel76] R. M. Keller. “Formal verification of parallel programs”. In: *Communications of the ACM* 19.7 (1976), pp. 371–384 (cit. on p. 21).
- [KYZ⁺11] S. Khalek, G. Yang, L. Zhang, D. Marinov, and S. Khurshid. “Testera: A tool for testing Java programs using Alloy specifications”. In: *ASE*. IEEE Computer Society, 2011, pp. 608–611 (cit. on p. 31).
- [KvdSV09] P. Klint, T. van der Storm, and J. Vinju. “RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation”. In: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009, pp. 168–177 (cit. on pp. 12, 16, 54, 102, 117, 131, 140, 144, 169).
- [Kli15] P. Klint. TypePal: Name and Type Analysis made Easy. <https://docs.rascal-mpl.org/unstable/TypePal/>. Accessed: 2022-01-15 (cit. on p. 142).
- [KRB18] J. Kubelka, R. Robbes, and A. Bergel. “The Road to Live Programming: Insights from the Practice”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: ACM, 2018, pp. 1090–1101. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180200 (cit. on p. 65).
- [Kum14] A. Kumar. “A lightweight formal approach for analyzing security of web protocols”. In: *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 192–211 (cit. on p. 3).
- [Lam99] L. Lamport. “Specifying concurrent systems with TLA+”. In: *Calculational System Design* (1999), pp. 183–247 (cit. on p. 5).
- [Lam02] L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002 (cit. on p. 5).
- [Lam00] A. v. Lamswerde. “Formal specification: a roadmap”. In: *Proceedings of the Conference on the Future of Software Engineering*. 2000, pp. 147–159 (cit. on p. 2).
- [LP10] D. Le Berre and A. Parrain. “The Sat4j library, release 2.2”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 7.2-3 (2010), pp. 59–64 (cit. on p. 9).

- [Leh96] M. M. Lehman. “Laws of software evolution revisited”. In: *European Workshop on Software Process Technology*. Springer. 1996, pp. 108–124 (cit. on p. 1).
- [Leh80] M. M. Lehman. “Programs, life cycles, and laws of software evolution”. In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076 (cit. on p. 1).
- [LSFo3] T. C. Lethbridge, J. Singer, and A. Forward. “How software engineers use documentation: The state of the practice”. In: *IEEE software* 20.6 (2003), pp. 35–39 (cit. on p. 2).
- [LMW20] M. Leuschel, M. Mutz, and M. Werth. “Modelling and Validating an Automotive System in Classical B and Event-B”. In: *International Conference on Rigorous State-Based Methods*. Springer. 2020, pp. 335–350 (cit. on pp. 109, 110).
- [LF95] H. Lieberman and C. Fry. “Bridging the gulf between code and behavior in programming”. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Press/Addison-Wesley Publishing Co. 1995, pp. 480–486 (cit. on p. 65).
- [LSS⁺15] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. “In defense of soundness: a manifesto”. In: *Communications of the ACM* 58.2 (2015), pp. 44–46 (cit. on p. 111).
- [MBC⁺16] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg. “Lightweight specification and analysis of dynamic systems with rich configurations”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 373–383 (cit. on pp. 102, 113).
- [MC13] N. Macedo and A. Cunha. “Implementing QVT-R Bidirectional Model Transformations Using Alloy”. In: *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 297–311. DOI: 10.1007/978-3-642-37057-1_22 (cit. on p. 85).
- [MGC13] N. Macedo, T. Guimarães, and A. Cunha. “Model repair and transformation with Echo”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. Ed. by E. Denney, T. Bultan, and A. Zeller. IEEE, 2013, pp. 694–697. DOI: 10.1109/ASE.2013.6693135 (cit. on pp. 66, 85).

- [MFCoo] T. Mackinnon, S. Freeman, and P. Craig. “Endo-testing: unit testing with mock objects”. In: *Extreme programming examined* (2000), pp. 287–301 (cit. on pp. 92, 111, 149).
- [MFL20] A. Mammar, M. Frappier, and R. Laleau. “An Event-B Model of an Automotive Adaptive Exterior Light System”. In: *International Conference on Rigorous State-Based Methods*. Springer. 2020, pp. 351–366 (cit. on pp. 109, 110).
- [MLK12] T. Mayerhofer, P. Langer, and G. Kappel. “A runtime model for fUML”. In: *Proceedings of the 7th Workshop on Models@ run. time*. ACM. 2012, pp. 53–58 (cit. on p. 86).
- [McC94] W. McCune. *A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems*. Tech. rep. Technical report, Argonne National Laboratory, 1994 (cit. on p. 60).
- [McD13] S. McDirmid. “Usable live programming”. In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. 2013, pp. 53–62. DOI: 10.1145/2509578.2509585 (cit. on p. 65).
- [MRT⁺17] B. Meng, A. Reynolds, C. Tinelli, and C. Barrett. “Relational constraint solving in SMT”. In: *CAV*. Springer. 2017, pp. 148–165 (cit. on p. 61).
- [MHS05] M. Mernik, J. Heering, and A. M. Sloane. “When and how to develop domain-specific languages”. In: *ACM computing surveys (CSUR)* 37.4 (2005), pp. 316–344 (cit. on p. 148).
- [Mey85] B. Meyer. “On Formalism in Specifications”. In: *IEEE Software* 2.1 (1985), pp. 6–26 (cit. on p. 16).
- [MJ14] A. Milicevic and D. Jackson. “Preventing arithmetic overflows in Alloy”. In: *Science of Computer Programming* 94 (2014), pp. 203–216 (cit. on p. 54).
- [MK11] A. Milicevic and H. Kugler. “Model checking using SMT and theory of lists”. In: *NASA Formal Methods*. Springer, 2011, pp. 282–297 (cit. on p. 22).
- [MNK⁺15] A. Milicevic, J. Near, E. Kang, and D. Jackson. “Alloy*: a general-purpose higher-order relational constraint solver”. In: *ICSE*. IEEE Press. 2015, pp. 609–619 (cit. on p. 54).
- [MQo8] M. Musuvathi and S. Qadeer. “Fair stateless model checking”. In: *ACM SIGPLAN Notices* 43.6 (2008), pp. 362–371 (cit. on p. 6).
- [NDD⁺17] T. Nelson, N. Danas, D. Dougherty, and S. Krishnamurthi. “The power of why and why not: enriching scenario exploration with provenance”. In: *FSE*. ACM. 2017, pp. 106–116 (cit. on p. 151).

- [NRZ⁺15] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. “How Amazon web services uses formal methods”. In: *Communications of the ACM* 58.4 (2015), pp. 66–73 (cit. on pp. 5, 6).
- [PSK⁺11] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. “Formal verification of OAuth 2.0 using Alloy framework”. In: *2011 International Conference on Communication Systems and Network Technologies*. IEEE. 2011, pp. 655–659 (cit. on p. 5).
- [Pet14] C. Peters. “FORS - Separating Configuration From Formal Specification”. MA thesis. University of Amsterdam, 2014 (cit. on p. 27).
- [Pla] PlantUML. PlantUML. <https://plantuml.com/state-diagram>. Accessed: 2022-01-09 (cit. on p. 139).
- [Plo81] G. D. Plotkin. *A structural approach to operational semantics*. Tech. rep. Computer Science Dept., Aarhus University, Denmark, 1981 (cit. on p. 21).
- [PFL17] C. Prud’homme, J. Fages, and X. Lorca. *Choco Documentation*. 2017. URL: <http://www.choco-solver.org> (cit. on p. 52).
- [RVV08] I. Ráth, D. Vago, and D. Varró. “Design-time simulation of domain-specific models by incremental pattern matching”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008, Herrsching am Ammersee, Germany, 15-19 September 2008, Proceedings*. 2008, pp. 219–222. DOI: 10.1109/VLHCC.2008.4639089 (cit. on pp. 86, 87).
- [RCG⁺17] G. Regis, C. Cornejo, S. Gutiérrez Brida, M. Politano, F. Raverta, P. Ponzio, N. Aguirre, J. P. Galeotti, and M. Frias. “DynAlloy Analyzer: A tool for the specification and analysis of Alloy models with dynamic behaviour”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 969–973 (cit. on p. 113).
- [RRL⁺19] P. Rein, S. Ramson, J. Lincke, R. Hirschfeld, and T. Pape. “Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives on Liveness”. In: *Programming Journal* 3.1 (2019), p. 1. DOI: 10.22152/programming-journal.org/2019/3/1 (cit. on p. 65).
- [RTG⁺13] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. “Finite model finding in SMT”. In: *CAV*. Springer. 2013, pp. 640–655 (cit. on p. 61).
- [RCB02] D. Richard, K. R. Chandramouli, and R. W. Butler. “Cost effective use of formal methods in verification and validation”. In: (2002) (cit. on p. 3).
- [Run06] P. Runeson. “A survey of unit testing practices”. In: *IEEE software* 23.4 (2006), pp. 22–29 (cit. on p. 113).
- [Sag15] S. Saghafi. “A Framework for Exploring Finite Models”. PhD thesis. Worcester Polytechnic Institute, 2015 (cit. on p. 60).

- [Sch94] A. Schürr. "Specification of Graph Translators with Triple Graph Grammars". In: *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*. 1994, pp. 151–163. DOI: 10.1007/3-540-59071-4_45 (cit. on p. 85).
- [ST15] R. Sebastiani and P. Trentin. "OptiMathSAT: a tool for optimization modulo theories". In: *CAV*. Springer. 2015, pp. 447–454 (cit. on p. 42).
- [SDH⁺16] O. Semeráth, C. Debreceňi, Á. Horváth, and D. Varró. "Incremental backward change propagation of view models by logic solvers". In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*. 2016, pp. 306–316 (cit. on p. 85).
- [Sim06] A. Simpson. "Logic, damned logic, and statistics". In: *Teaching Formal Methods: Practice and Experience* (2006), pp. 1–6 (cit. on p. 5).
- [Sla94] J. Slaney. "FINDER: Finite domain enumerator system description". In: *CAV*. Springer. 1994, pp. 798–801 (cit. on p. 60).
- [SD98] M. Snoeck and G. Dedene. "Existence Dependency: The key to semantic integrity between structural and behavioral aspects of object types". In: *Software Engineering, IEEE Transactions on* 24.4 (1998), pp. 233–251 (cit. on p. 28).
- [Sno14] M. Snoeck. *Enterprise Information Systems Engineering*. Springer, 2014 (cit. on p. 6).
- [SMD03] M. Snoeck, C. Michiels, and G. Dedene. "Consistency by construction: the case of MERODE". In: *International Conference on Conceptual Modeling*. Springer. 2003, pp. 105–117 (cit. on pp. 6, 18, 24, 28).
- [SSV20] T. Soethout, T. v. d. Storm, and J. J. Vinju. "Automated Validation of State-Based Client-Centric Isolation with TLA+". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2020, pp. 43–57 (cit. on p. 152).
- [SvdSV21a] T. Soethout, T. van der Storm, and J. J. Vinju. "Path-Sensitive Atomic Commit-Local Coordination Avoidance for Distributed Transactions." In: *The Art, Science, and Engineering of Programming* 5.1 (2021), p. 3 (cit. on p. 152).
- [SAB⁺17] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli. "To mock or not to mock? an empirical study on mocking practices". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 402–412 (cit. on pp. 111, 149).
- [Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. USA: Prentice-Hall, Inc., 1989. ISBN: 013983768X (cit. on p. 3).

- [SHU16] F. Steimann, J. Hagemann, and B. Ulke. “Computing Repair Alternatives for Malformed Programs Using Constraint Attribute Grammars”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: ACM, 2016, pp. 711–730. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2984007 (cit. on p. 86).
- [Sto16] J. Stoel. *Rebel*. 2016. DOI: unknown. URL: <https://github.com/cwi-swat/rebel> (cit. on p. 12).
- [Sto19a] J. Stoel. *AlleAlle*. 2019. DOI: unknown. URL: <https://github.com/cwi-swat/allealle> (cit. on p. 12).
- [Sto19b] J. Stoel. *AlleAlle Benchmarks*. 2019. DOI: unknown. URL: <https://github.com/joukestoel/allealle-benchmark> (cit. on p. 12).
- [Sto19c] J. Stoel. *Nextep Live Statemachine Example*. 2019. DOI: unknown. URL: <https://github.com/joukestoel/live-state-machines> (cit. on p. 12).
- [Sto21] J. Stoel. *Rebel2*. 2021. DOI: unknown. URL: <https://github.com/cwi-swat/rebel2> (cit. on p. 12).
- [SSV⁺16] J. Stoel, T. v. d. Storm, J. Vinju, and J. Bosman. “Solving the bank with Rebel: on the design of the Rebel specification language and its application inside a bank”. In: *Proceedings of the 1st Industry Track on Software Language Engineering*. 2016, pp. 13–20 (cit. on pp. 10, 93).
- [STvdS⁺19] J. Stoel, U. Tikhonova, T. van der Storm, and T. Degueule. *Nextep*. 2019. DOI: unknown. URL: <https://github.com/cwi-swat/live-modeling> (cit. on p. 12).
- [SvdSV19] J. Stoel, T. van der Storm, and J. J. Vinju. “AlleAlle: bounded relational model finding with unbounded data”. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 2019, pp. 46–61 (cit. on pp. 10, 92, 102, 106).
- [SvdSV21b] J. Stoel, T. van der Storm, and J. J. Vinju. “Modeling with Mocking”. In: *2021 IEEE 14th International Conference on Software Testing, Validation and Verification (ICST)*. 2021 (cit. on p. 11).
- [SPM11] R. V. D. Straeten, J. P. Puissant, and T. Mens. “Assessing the Kodkod Model Finder for Resolving Model Inconsistencies”. In: *Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011 Proceedings*. Ed. by R. B. France, J. M. Küster, B. Bordbar, and R. F. Paige. Vol. 6698. Lecture Notes in Computer Science. Springer, 2011, pp. 69–84. DOI: 10.1007/978-3-642-21470-7_6 (cit. on p. 86).

- [Tan13] S. L. Tanimoto. “A perspective on the evolution of live programming”. In: *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*. Ed. by B. Burg, A. Kuhn, and C. Parnin. IEEE Computer Society, 2013, pp. 31–34. DOI: 10.1109/LIVE.2013.6617346 (cit. on p. 65).
- [TSV⁺18] U. Tikhonova, J. Stoel, T. Van Der Storm, and T. Degueule. “Constraint-based run-time state migration for live modeling”. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. 2018, pp. 108–120 (cit. on p. 10).
- [TP04] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code (Monographs in Computer Science)*. Springer, 2004 (cit. on p. 17).
- [Tor09] E. Torlak. “A Constraint Solver for Software Engineering : Finding Models and Cores of Large Relational Specifications”. PhD thesis. Massachusetts Institute of Technology, 2009 (cit. on pp. 33, 52, 53, 55, 151).
- [TB13] E. Torlak and R. Bodik. “Growing solver-aided languages with Rosette”. In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software* (2013), pp. 135–152 (cit. on p. 151).
- [TB14] E. Torlak and R. Bodik. “A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation 2* (2014), pp. 530–541. ISSN: 15232867 (cit. on p. 118).
- [TC]08] E. Torlak, F. Chang, and D. Jackson. “Finding minimal unsatisfiable cores of declarative specifications”. In: *FM*. Springer. 2008, pp. 326–341 (cit. on p. 62).
- [T]07a] E. Torlak and D. Jackson. “Kodkod: A relational model finder”. In: *TACAS*. Springer. 2007, pp. 632–647 (cit. on pp. 33, 47, 60, 62).
- [TD06] E. Torlak and G. Dennis. “Kodkod for Alloy users”. In: *First ACM Alloy Workshop, Portland, Oregon*. ACM, 2006 (cit. on pp. 27, 33, 60).
- [T]07b] E. Torlak and D. Jackson. “Kodkod: A relational model finder”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2007, pp. 632–647 (cit. on pp. 5, 9, 35, 106, 113, 119, 149, 151, 237).
- [TSJ⁺07] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. “Opium: Optimal package install/uninstall manager”. In: *ICSE*. IEEE Computer Society. 2007, pp. 178–188 (cit. on p. 57).

- [VD16] A. Vakili and N. Day. “Finite Model Finding Using the Logic of Equality with Uninterpreted Functions”. In: *FM*. Springer. 2016, pp. 677–693 (cit. on p. 60).
- [vdSto13] T. van der Storm. “Semantic deltas for live DSL environments”. In: *1st International Workshop on Live Programming (LIVE)*. IEEE. 2013, pp. 35–38 (cit. on p. 65).
- [vGas16] R. van Gasteren. “Natural Language Generation from Rebel Specifications”. MA thesis. Utrecht, the Netherlands: University of Utrecht, 2016 (cit. on p. 148).
- [vMvTV17] S. van Mierlo, Y. van Tendeloo, and H. Vangheluwe. “Debugging Parallel DEVS”. In: *Simulation* 93.4 (2017), pp. 285–306. DOI: 10.1177/0037549716658360 (cit. on p. 86).
- [vRvdS17] R. van Rozen and T. van der Storm. “Toward live domain-specific languages”. In: *Software & Systems Modeling* (Aug. 2017) (cit. on pp. 65, 78, 83).
- [VB07] D. Varró and A. Balogh. “The model transformation language of the VIATRA2 framework”. In: *Sci. Comput. Program.* 68.3 (2007), pp. 214–234. DOI: 10.1016/j.scico.2007.05.004 (cit. on p. 87).
- [VBG⁺09] M. Veanes, N. Bjørner, Y. Gurevich, and W. Schulte. “Symbolic bounded model checking of abstract state machines”. In: *Int J Software Informatics* 3 (2009), pp. 149–170 (cit. on p. 22).
- [Ver15] S. Verweij. *State Machine cat*. <https://github.com/sverweij/state-machine-cat>. Accessed: 2022-01-15 (cit. on p. 143).
- [ZSR⁺18] A. Zamansky, M. Spichkova, G. Rodriguez-Navas, P. Herrmann, and J. Blech. “Towards classification of lightweight formal methods”. In: *13th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2018, 23 March 2018 through 24 March 2018*. SciTePress. 2018, pp. 305–313 (cit. on p. 5).
- [Zav17] P. Zave. “Reasoning about identifier spaces: How to make chord correct”. In: *IEEE Transactions on Software Engineering* 43.12 (2017), pp. 1144–1156 (cit. on p. 5).
- [ZZ95] J. Zhang and H. Zhang. “SEM: a system for enumerating models”. In: *IJCAI*. Vol. 95. 1995, pp. 298–303 (cit. on p. 60).

ABOUT RASCAL

a.1 RASCAL PROGRAMS

Rascal is a functional/procedural programming language with immutable data, builtin generic containers (lists, sets, relations, tuples, etc.), abstract and concrete syntax declarations, high-level polymorphic pattern matching, traversal, substitution and query primitives. The high-level features are meant to cover what is needed for language design and implementation, as well as reverse engineering from code. Pattern matching and substitution, generic traversal and relational calculus are the core features of the language that find different applications. For example, "patterns" drive dynamic dispatch of overloaded functions, and bind the generic traversal 'visit' statement to specific ad-hoc types at run-time, and produce the different bindings to iterate over for comprehensions. Function bodies can be either expressions or simply structured programming basic blocks (assignment, 'if-then-else', 'for', 'while', 'try-catch'), including lexically scoped (re)assignment of variables and lexically scoped backtracking. Choice points for backtracking in Rascal are produced by either generators (of elements of containers) or alternative patterns, or non-unitary patterns. There is no 'null' or 'uninitialized' in Rascal, since all binding via pattern matching with conditional scopes, and all optional values have defaults. Further information can be found at <https://www.rascal-impl.org/docs> [KvdSV09].

a.2 RASCAL SYNTAX DEFINITIONS

Rascal is a meta programming language with embedded context-free grammars in a BNF-like notation. `REBEL`, `ALLEALLE`, `NEXTEP` and `REBEL2` are defined in Rascal notation in the comming appendices. We summarize the Rascal notation here.

- Non-terminals act as algebraic data-types: **syntax** `Exp = Exp "+" Exp` defines the type `Exp` for parse trees generated for the `Exp` non-terminal.
- There is no separate scanner or lexer; all syntax is defined with context-free grammar rules: **lexical** `Id = [a-z]+` defines identifiers.
- Whitespace and comment non-terminals are interspersed with all **syntax** rules: `layout W = []` mixes into the aforementioned `Exp` definition like so **syntax** `Exp = Exp W "+" W Exp`.
- Regular expressions over non-terminals (a.k.a. extended BNF) allow for shortcuts for lists and separated lists: `{Stat ";"}` is a list of `Stat` separated by semicolons.
- Character classes and character class operations are used to specify "lexical" syntax: `[A-Za-z0-9]` a class of (capital) roman letters including the digits.

- Literal tokens and case-insensitive literal tokens are used for keywords and punctuation: i.e., "if" and the case-insensitive variant 'if'.
- A partial order between mutually recursive production rules declares binding strength: `Exp "*" Exp > Exp "+" Exp`.
- Associativity 'left' and 'right' declares binding strength to the left of right of a recursive operator: `left Exp "+" Exp`.
- A language difference operator `Id \ Keywords` generates the difference between the language of `Id` and `Exp`.
- A follow restriction operator `Id !>> [a-z]` declares that `Id` instances can not be followed immediately by any character in `[a-z]`.
- A rule can be tagged with a constructor name and field names for the elements of the rule: `Exp = addition: Exp lhs "+" Exp rhs`

There are no restrictions on the shape of the grammar: even left-recursion is allowed, and non-determinism or ambiguity as well. From the grammar definition Rascal generates a context-free general parsing algorithm based on GLL.

SYNTAX DEFINITIONS

b.1 SYNTAX DEFINITION OF REBEL

```

1  start syntax Module
2  = ModuleDeflexicals modDef Import* imports Specification spec
3  | ModuleDef modDef Import* imports LibraryModule* decls;
4
5  syntax ModuleDef = "module" FullyQualifiedName fqName;
6
7  syntax FullyQualifiedName = ({VarName "."}+ packages ".")? modulePath TypeName modName;
8  syntax FullyQualifiedVarName = (FullyQualifiedName fqName ".")? VarName name;
9
10 syntax Import = "import" FullyQualifiedName fqName;
11
12 syntax Expr
13 = bracket "(" Expr ")"
14 > literal: Literal!reference lit
15 | reference: Ref ref
16 | VarName function "(" {Expr ","}* exprs ")"
17 | left fieldAccess: Expr lhs "." VarName field
18 | "{" Expr lower "." Expr upper}"
19 | Expr var!accessor "[" Expr indx "]"
20 | "(" {MapElement ","}* mapElems ")"
21 | staticSet: "{" {Expr ","}* setElems}"
22 | comprehension: "{" VarName elemName ":" Expr set "|" {Expr ","}* conditions}"
23 | cardanality: "|" Expr set "|"
24 | universalQuantifier: "forall" VarName elemName ":" Expr set "|" {Expr ","}* conditions
25 | existentialQuantifier: "exists" VarName elemName ":" Expr set "|" {Expr ","}* conditions
26 > new: "new" Expr expr
27 | "not" Expr expr
28 | "-" Expr
29 > Expr cond "?" Expr whenTrue ":" Expr whenFalse
30 > left ( Expr lhs "*" Expr rhs
31 | isMember: Expr lhs "in" Expr rhs
32 | Expr lhs "/" Expr rhs
33 | Expr lhs "%" Expr rhs
34 )
35 > left ( Expr lhs "+" Expr rhs
36 | subtract: Expr lhs "-" Expr rhs
37 )
38 > non-assoc ( smallerThan: Expr lhs "<" Expr rhs
39 | smallerThanEquals: Expr lhs "<=" Expr rhs
40 | greaterThan: Expr lhs ">" Expr rhs
41 | greaterThanEquals: Expr lhs ">=" Expr rhs
42 | equals: Expr lhs "==" Expr rhs
43 | notEqual: Expr lhs "!=" Expr rhs
44 )
45 > "initialized" Expr
46 | "finalized" Expr
47 | Expr lhs "instate" StateRef sr
48 > left and: Expr lhs "&&" Expr rhs
49 > left Expr lhs "||" Expr rhs
50 | right Expr cond "->" Expr implication;
51
52 syntax StateRef
53 = VarName state
54 | "{" VarName+ states}";
55

```

```

56 syntax MapElement = Expr key ":" Expr val;
57
58 syntax Ref
59   = FullyQualifiedVarName field
60   | FullyQualifiedName tipe
61   | this: "this"
62   | "it";
63
64 syntax Type
65   = "Boolean"
66   | "Period"
67   | "Integer"
68   | "Money"
69   | "Currency"
70   | "Date"
71   | "Frequency"
72   | "Percentage"
73   | "Period"
74   | "Term"
75   | "String"
76   | "map" "[" Type ":" Type "]"
77   | "set" "[" Type "]"
78   | Term
79   | "Time"
80   | "IBAN"
81   | "InterestNorm"
82   | "InterestTariff"
83   | Type "-\>" Type
84   | "{" Type ","+ "}"
85   | TypeName custom;
86
87 syntax Literal
88   = Int
89   | Bool
90   | Period
91   | Frequency
92   | Term term
93   | Date
94   | Time
95   | DateTime
96   | Percentage
97   | String
98   | Money
99   | Currency
100  | IBAN
101  | InterestNorm
102  | InterestTariff;
103
104 syntax Literal = anyVal: "ANY";
105 syntax Date = Int day Month month Int? year;
106
107 syntax Time
108   = hhmm: [0-9][0-9]? hour ":" [0-9][0-9]? minutes
109   | hhmmss: [0-9][0-9]? hour ":" [0-9][0-9]? minutes ":" [0-9][0-9]? seconds;
110
111 syntax DateTime
112   = Date date "," Time time
113   | "now";
114
115 syntax Term = Int factor Period period;
116 syntax Money = Currency cur MoneyAmount amount;
117
118 syntax InterestTariff
119   = fixed:          "fixed" Percentage p

```

```

120 | normBased:      "norm" InterestNorm norm
121 | withPosAddition: InterestTariff base "+" InterestTariff posAddition
122 | withNegAddition: InterestTariff base "--" InterestTariff negAddition
123 | withMinMaxTariff: InterestTariff base "bounded by" ("min" "=" InterestTariff min)?
124 | ("max" "=" InterestTariff max)?;
125
126 syntax LibraryModule
127 = EventDef eventDef
128 | FunctionDef functionDef
129 | InvariantDef invariantDef;
130
131 // Library rules
132
133 syntax EventDef = Annotations annos "event" FullyQualifiedVarName name EventConfigBlock? configParams
134 | (" {Parameter ", "}* transitionParams") " {"
135 | Preconditions? pre Postconditions? post MaybeSyncBlock sync
136 | "};";
137
138 syntax MaybeSyncBlock = SyncBlock?;
139 syntax EventConfigBlock = "[" {Parameter ", ")+ params "];";
140 syntax Preconditions = "preconditions" "{" Statement* stats "};";
141 syntax Postconditions = "postconditions" "{" Statement* stats "};";
142 syntax SyncBlock = "sync" "{" SyncStatement* stats "};";
143
144 syntax FunctionDef = Annotations annos "function" FullyQualifiedVarName name
145 | (" {Parameter ", "}* params ") " ":" Type returnType "=" Statement statement;
146
147 syntax InvariantDef = Annotations annos "invariant" FullyQualifiedVarName name
148 | "{" Statement* stats "};";
149
150 // Specification rules
151
152 syntax Specification =
153 | spec:Annotations annos SpecModifier? modifier "specification" TypeName name Extend? extend "{"
154 | Fields? optFields EventRefs? optEventRefs InvariantRefs? optInvariantRefs LifeCycle? optLifeCycle
155 | "};";
156
157 syntax Extend = "extends" FullyQualifiedName parent;
158 syntax Fields = @Foldable "fields" "{" FieldDecl* fields "};";
159 syntax EventRefs = @Foldable eventInstances: "events" "{" EventRef* events "};";
160
161 syntax EventRef
162 = ref: FullyQualifiedVarName eventRef "[" {ConfigParameter ", "}* config "]"
163 | interfaceDecl: VarName name "(" Parameter ", "}* params ");";
164
165 syntax InvariantRefs = @Foldable "invariants" "{" FullyQualifiedVarName* invariants "};";
166 syntax LifeCycle = "lifeCycle" "{" StateFrom* from "};";
167
168 syntax StateFrom = LifeCycleModifier? mod VarName from StateTo* destinations;
169 syntax StateTo = "->" VarName to ":" StateVia via;
170 syntax StateVia = {VarName ", ")+ refs;
171
172 // Generic rules
173 syntax ConfigParameter = VarName name "=" Expr val;
174 syntax Parameter = VarName name ":" Type tipe DefaultValue? defaultValue;
175 syntax DefaultValue = "=" Expr val;
176 syntax SyncStatement = Annotations doc SyncExpr expr ";";
177
178 syntax SyncExpr
179 = not: "not" SyncExpr expr
180 | syncEvent: TypeName specName "[" Expr id "]" "." VarName event "(" {Expr ", "}* params ");";
181
182 syntax Statement
183 = bracket "(" Statement ")"

```

```

184 | "case" Expr "{" Case+ cases "}" ";";
185 | Annotations annos Expr expr ";";
186
187 syntax Case = Literal lit "=>" Statement stat;
188
189 syntax StateRef
190 = VarName state
191 | "{" VarName+ states "}";
192
193 syntax MapElement = Expr key ":" Expr val;
194
195 syntax FieldDecl = VarName name ":" Type tipe Annotations meta;
196
197 syntax Ref
198 = FullyQualifiedVarName field
199 | FullyQualifiedName tipe
200 | this: "this"
201 | "it";
202
203 syntax Annotations = Annotation* annos;
204
205 syntax Annotation
206 = key: "@" "key"
207 | ref: "@" "ref" "=" FullyQualifiedName spc
208 | doc: "@" VarName name TagString tagString;
209
210 lexical InterestNorm = "EURIBOR" | "AIRBOR" | "LIBOR" | "INGBASIS" | "LIMITBASED";
211
212 lexical Currency
213 = "EUR" | "USD"
214 | "CUR" '_' ([A-Z][A-Z][A-Z]) name;
215
216 lexical IBAN = [A-Z] !<<
217 ([A-Z][A-Z]) countryCode ([0-9][0-9]) checksum [0-9 A-Z]+ accountNumber
218 !>> [0-9 A-Z];
219
220 lexical TypeName = ([A-Z] !<< [A-Z][a-z 0-9 _][a-z A-Z 0-9 _]* !>> [a-z A-Z 0-9 _]) \Keywords;
221 lexical VarName = ([a-z] !<< [a-z][a-z A-Z 0-9 _]* !>> [a-z A-Z 0-9 _]) \Keywords;
222 lexical Month = "Jan" | "Feb" | "Mar" | "Apr" | "May" | "Jun" |
223 "Jul" | "Aug" | "Sep" | "Oct" | "Nov" | "Dec";
224 lexical Frequency = "Daily" | "Weekly" | "Monthly" | "Quarterly" | "Yearly";
225 lexical Period = "Day" | "Week" | "Month" | "Quarter" | "Year";
226 lexical Bool = "True" | "False";
227 lexical Percentage = [0-9]+ per "%";
228 lexical Int = [0-9]+ | "Inf";
229 lexical String = "\"" ![\"]* "\"";
230 lexical MoneyAmount = [0-9]+ whole [.] ([0-9][0-9][0-9]?) decimals;

```

Listing B.1: Rascal definition of REBEL syntax.

b.2 SYNTAX DEFINITION OF ALLEALLE

```

1  start syntax Problem = problem: Relation* relations AlleConstraint* constraints
2     ObjectiveSection? objSection Expect? expect;
3
4  syntax Relation = RelVar v "(" {HeaderAttribute ","}* header ")" RelationalBound bounds;
5
6  syntax HeaderAttribute = AttributeName name ":" Domain dom;
7
8  syntax AttributeHeader = header: AttributeName name ":" Domain dom;
9
10 syntax RelationalBound
11 = exact: "=" {" {Tuple ","}*tuples "}
12 | atMost: "\<=" {" {Tuple ","}* upper "}
13 | atLeastAtMost: "\>=" {" {Tuple ","}* lower "} "\<=" {" {Tuple ","}* upper "}";
14
15 syntax Tuple
16 = tup: "\<" {Value ","}* values "\>"
17 | range: "\<" {RangedValue ","}* from "\>" ".." "\<" {RangedValue ","}* to "\>";
18
19 syntax Value
20 = Idd id
21 | lit: Literal lit
22 | "?";
23
24 syntax RangedValue
25 = id: RangedId prefix RangedNr numm
26 | idOnly: RangedId id
27 | templateLit: Literal lit
28 | "?";
29
30 syntax Domain = "id";
31
32 syntax Literal = idLit: "\'" Idd id "\'";
33
34 syntax AlleConstraint
35 = AlleFormula form
36 | AllePredicate predDef;
37
38 syntax AllePredicate = "pred" Idd name "[" {PredParam ","}* params "]" "=" AlleFormula form;
39
40 syntax PredParam = RelVar name ":" "(" {HeaderAttribute ","}* header ")";
41
42 syntax AlleFormula
43 = bracket "(" AlleFormula form ")"
44 > predCall: Idd predName "[" {AlleExpr ","}* args "]"
45 > negation: "not" AlleFormula form
46 > empty: "no" AlleExpr expr
47 | atMostOne: "lone" AlleExpr expr
48 | exactlyOne: "one" AlleExpr expr
49 | nonEmpty: "some" AlleExpr expr
50 | subset: AlleExpr lhsExpr "in" AlleExpr rhsExpr
51 | left equal: AlleExpr lhsExpr "=" AlleExpr rhsExpr
52 | left unequal: AlleExpr lhsExpr "!=" AlleExpr rhsExpr
53 > left conjunction: AlleFormula lhsForm "&&" AlleFormula rhsForm
54 | left disjunction: AlleFormula lhsForm "||" AlleFormula rhsForm
55 > implication: AlleFormula lhsForm "=>" AlleFormula rhsForm
56 | equality: AlleFormula lhsForm "\<=>" AlleFormula rhsForm
57 > let: "let" {VarBinding ","}* bindings "|" AlleFormula form
58 > universal: "forall" {VarDeclaration ","}* decls "|" AlleFormula form
59 | existential: "exists" {VarDeclaration ","}* decls "|" AlleFormula form;
60
61 syntax VarDeclaration = varDecl: RelVar var ":" AlleExpr expr;

```

```

62
63 syntax VarBinding = varBinding: RelVar var "=" AlleExpr expr;
64
65 syntax AlleExpr
66 = bracket "(" AlleExpr expr ")"
67 > variable: RelVar v
68 | lit: Literal l
69 > rename: AlleExpr r "[" {Rename ","}+ "]"
70 | project: AlleExpr r "[" {AttributeName ","}+ "]"
71 | renameAndProject: AlleExpr r "[" {ProjectAndRename ","}+ "]"
72 | select: AlleExpr r "where" Criteria criteria
73 | aggregate: AlleExpr r "[" {AggregateFunctionDef ","}+ "]"
74 | groupedAggregate: AlleExpr r "[" {AttributeName ","}+ groupBy ","
75 {AggregateFunctionDef ","}+ aggregateFunctions "]"
76 > transpose: "~" AlleExpr r
77 | closure: "^" AlleExpr r
78 | reflexClosure: "*" AlleExpr r
79 > left naturalJoin: AlleExpr lhs "|x|" AlleExpr rhs
80 > left (union: AlleExpr lhs "+" AlleExpr rhs
81 |intersection: AlleExpr lhs "&" AlleExpr rhs
82 |difference: AlleExpr lhs "-" AlleExpr rhs
83 |product: AlleExpr lhs "x" AlleExpr rhs
84 )
85 | comprehension: "{" {VarDeclaration ","}+ decls "|" AlleFormula form "}"
86 ;
87
88 syntax TupleAttributeSelection
89 = "\<" AttributeName first "," AttributeName second "\>";
90
91 syntax Rename = AttributeName orig "as" AttributeName new;
92
93 syntax ProjectAndRename = AttributeName orig "-\>" AttributeName new;
94
95 syntax AggregateFunctionDef
96 = AggregateFunction func
97 | AggregateFunction func "as" AttributeName bindTo;
98
99 syntax AggregateFunction
100 = dummy: " " !>> " ";
101
102 syntax Criteria
103 = bracket "(" Criteria ")"
104 > "not" Criteria
105 > non-assoc
106 ( CriteriaExpr lhsExpr "=" CriteriaExpr rhsExpr
107 | CriteriaExpr lhsExpr "!=" CriteriaExpr rhsExpr
108 )
109 > left ( Criteria lhs "&&" Criteria rhs
110 | Criteria lhs "||" Criteria rhs
111 );
112
113 syntax CriteriaExpr
114 = bracket "(" CriteriaExpr ")"
115 | AttributeName att
116 | Literal l
117 > left Criteria condition "?" CriteriaExpr thn ":" CriteriaExpr els;
118
119 syntax ObjectiveSection
120 = "objectives" ":" {Objective ","}+ objectives
121 | "objectives" "(" ObjectivePriority prio ")" ":" {Objective ","}+ objectives;
122
123
124 syntax ObjectivePriority
125 = "lex"

```



```

126     | "pareto"
127     | "independent";
128
129 syntax Objective
130   = maximize: "maximize" AlleExpr expr
131     | minimize: "minimize" AlleExpr expr;
132
133 syntax Expect
134   = "expect" ":" ResultType result ("," ModelRestriction models)?;
135
136 syntax ResultType
137   = "sat"
138     | "trivial-sat"
139     | "unsat"
140     | "trivial-unsat";
141
142 syntax ModelRestriction
143   = "#" "models" ("(" Domain dom ")")? ModelRestrExpr expr;
144
145 syntax ModelRestrExpr
146   = "=" Arity
147     | "\>" Arity
148     | "\<" Arity;
149
150 lexical RangedId = ([a-zA-Z] !<< [a-zA-Z][a-zA-Z\-.]* !>> [a-zA-Z\-.]) \ Keywords;
151 lexical RangedNr = [0-9]+;
152 lexical Idd = ([a-zA-Z] !<< [a-zA-Z][a-zA-Z\0-9]* !>> [a-zA-Z\0-9]) \ Keywords;
153 lexical AttributeName = ([a-zA-Z] !<< [a-zA-Z][a-zA-Z0-9\_']* !>> [a-zA-Z0-9_]) \ Keywords;
154 lexical Arity = [0-9]+;
155 lexical RelVar = ([a-zA-Z] !<< [a-zA-Z][a-zA-Z0-9\_']* !>> [a-zA-Z0-9_]) \ Keywords;

```

Listing B.2: Syntax definition of the common constructs of ALLEALLE in Rascal.

```

1 syntax Domain = "int";
2
3 syntax Value      = neglit: "-" Literal lit;
4 syntax RangedValue = neglit: "-" Literal lit;
5
6 syntax Criteria
7   = non-assoc (lt: CriteriaExpr lhsExpr "\<" CriteriaExpr rhsExpr
8     | lte: CriteriaExpr lhsExpr "\<=" CriteriaExpr rhsExpr
9     | gt: CriteriaExpr lhsExpr "\>" CriteriaExpr rhsExpr
10    | gte: CriteriaExpr lhsExpr "\>=" CriteriaExpr rhsExpr
11    );
12
13 syntax CriteriaExpr
14   = abs:      "|" CriteriaExpr expr "|"
15     | neg:    "-" CriteriaExpr expr
16     > left mult: CriteriaExpr lhs "*" CriteriaExpr rhs
17     | non-assoc ( div: CriteriaExpr lhs "/" CriteriaExpr rhs
18       | \mod: CriteriaExpr lhs "%" CriteriaExpr rhs
19       )
20     > left ( add: CriteriaExpr "+" CriteriaExpr rhs
21       | sub: CriteriaExpr "-" CriteriaExpr rhs
22       )
23     > non-assoc ( "min" "(" CriteriaExpr a ", " CriteriaExpr b ")")
24       | "max" "(" CriteriaExpr a ", " CriteriaExpr b ")")
25     );
26
27 syntax Literal = intLit: IntLit i;
28
29 syntax AggregateFunction
30   = car: "count" "(" )

```

```
31 | sum: "sum" "(" AttributeName att ")"  
32 | min: "min" "(" AttributeName att ")"  
33 | max: "max" "(" AttributeName att ")"  
34 | avg: "avg" "(" AttributeName att ");"  
35  
36 lexical IntLit = [0-9]+;
```

Listing B.3: Syntax definition of the integer operation of ALLEALLE in Rascal.

b.3 SYNTAX DEFINITION OF NEXTEP IN RASCAL

```
1  start syntax Spec = StaticDef static DynamicDef dynamic MigrationDef migration DistanceDef? distance;
2
3  syntax StaticDef = "static" "{" Class* classes "}";
4  syntax DynamicDef = "runtime" "{" Class* classes "}";
5  syntax MigrationDef = "migration" "{" Formula* rules "}";
6  syntax DistanceDef = "distance" "{" PriorityDistance* priorities "}";
7
8  syntax Class = "class" ClassName name "{" ClassBody body "}";
9  syntax ClassBody = FieldDecl* fields Invariant* inv;
10
11 syntax FieldDecl = VarName fieldName ":" Type type "*" ? ;
12
13 syntax Invariant
14   = "invariant" ":" Formula form
15   | "invariants" "{" Formula+ forms "}";
16
17 syntax Formula
18   = bracket "(" Formula ")"
19   > neg:      "not" Formula
20   > some:     "some" Expr
21   | no:      "no" Expr
22   | \one:    "one" Expr
23   > subset:  Expr "in" Expr
24   | equality: Expr "=" Expr
25   | inequality: Expr "!=" Expr
26   > implies: Formula "=>" Formula
27   | iff:     Formula "\<=>" Formula
28   > conj:    Formula "&&" Formula
29   | disj:   Formula "||" Formula
30   > forall:  "forall" {QuantDecl ","}+ decls "|" Formula form
31   | exists:  "exists" {QuantDecl ","}+ decls "|" Formula form;
32
33 syntax QuantDecl = VarName ":" Expr;
34
35 syntax Formula
36   = intGte:   Expr "\>=" Expr
37   | intGt:    Expr "\>" Expr
38   | intLte:   Expr "\<=" Expr
39   | intLt:    Expr "\<" Expr;
40
41 syntax Expr
42   = bracket  "(" Expr ")"
43   > var:     VarName
44   | lit:     Literal
45   | left dotJoin: Expr "." Expr
46   | left relJoin: Expr "\<->" Expr
47   > restrict: Expr "where" RestrictStat
48   > left ( union: Expr "++" Expr
49           | intersec: Expr "&" Expr
50           | setDif: Expr "--" Expr
51           )
52   > transCl:  "^" Expr
53   | reflTrans: "*" Expr
54   > old: "old" "[" Expr expr "]"
55   | new: "new" "[" Expr expr "]" ;
56
57 syntax Expr
58   = abs:     "|" Expr "|"
59   > left ( div: Expr "\\" Expr
60           | mul: Expr "*" Expr
61           > add: Expr "+" Expr
```

```

62         | sub: Expr "-" Expr
63         );
64
65 syntax RestrictStat = "(" RestrictExpr "=" RestrictExpr ")";
66
67 syntax RestrictExpr = QualifiedName att;
68
69 syntax QualifiedName = left VarName ( "." VarName)*;
70
71 syntax Literal = intLit: Int;
72
73 syntax PriorityDistance = Expr distance ":" Int priority;
74
75 syntax Type
76   = class: ClassName className
77   | \int: "int"
78   ;
79
80 lexical ClassName = ([A-Z] !<< [A-Z][a-zA-Z0-9_\'']* !>> [a-zA-Z0-9_]) \ Keywords;
81 lexical VarName = ([a-zA-Z] !<< [a-zA-Z][a-zA-Z0-9_\'']* !>> [a-zA-Z0-9_]) \ Keywords;
82 lexical Atom = ([a-zA-Z] !<< [a-zA-Z][a-zA-Z0-9_\'']* !>> [a-zA-Z0-9_]) \ Keywords;
83
84 lexical Int = [0-9]+;

```

Listing B.4: Rascal definition of NEXTEP syntax.

b.4 SYNTAX DEFINITION OF REBEL2

This appendix contains the Rascal syntax definition of REBEL2. It is split in the definitions of lexical constructs (Listing B.5), common syntax definitions (Listing B.6), the syntax definitions that can be used when defining configurations and checks (Listing B.7) and the syntax definitions that can be used when defining specifications (Listing B.8). All three parts together make up the complete syntax definition of REBEL2.

```
1 lexical Id = [a-z A-Z 0-9 _] !<< ([a-z A-Z][a-z A-Z 0-9 _])* !>> [a-z A-Z 0-9 _] \ Keywords;
2
3 lexical TypeName = [a-z A-Z 0-9 _] !<< [A-Z][a-z A-Z 0-9 _]* !>> [a-z A-Z 0-9 _] \ Keywords;
4
5 lexical Int = [0-9] !<< [0-9]+ !>> [0-9];
6
7 lexical StringConstant = "\"" StringCharacter* "\"";
8
9 lexical UnicodeEscape
10   = utf16: "\\\" [u] [0-9 A-F a-f] [0-9 A-F a-f] [0-9 A-F a-f] [0-9 A-F a-f]
11   | utf32: "\\\" [U] ((\"0\" [0-9 A-F a-f])|\"10\") [0-9 A-F a-f] [0-9 A-F a-f] [0-9 A-F a-f] [0-9 A-F a-f]
12   | ascii: "\\\" [a] [0-7] [0-9A-Fa-f];
13
14 lexical StringCharacter
15   = "\\\" [\" \' \< \> \\ b f n r t]
16   | UnicodeEscape
17   | ![\" \' \< \> \\]
18   | [\n] [\ \t \u00A0 \u1680 \u2000-\u200A \u202F \u205F \u3000]* [\'];
```

Listing B.5: Rascal definition of REBEL2 lexicals.

```

1  start syntax Module = ModuleId module Import* imports Part+ parts;
2
3  syntax Part = ; // Is extended in later definitions
4  syntax ModuleId = "module" QualifiedName name;
5  syntax Import = "import" QualifiedName module;
6  syntax QualifiedName = {Id "::."}+ names !> "::.";
7
8  syntax Formula
9    = brackets: "(" Formula ")"
10   > "!" Formula form
11   > sync: Expr spc "." QualifiedName event "(" {Expr ","}* params ")"
12   | inState: Expr expr "is" QualifiedName state
13   | membership: Expr "in" Expr
14   | nonMembership: Expr "notin" Expr
15   > Expr "\<" Expr
16   | Expr "\<=" Expr
17   | Expr "=" Expr
18   | Expr "!=" Expr
19   | Expr "\>=" Expr
20   | Expr "\>" Expr
21   > right Formula "&&" Formula
22   | right Formula "||" Formula
23   > right Formula "=\<>" Formula
24   | right Formula "\<=>" Formula
25   | non-assoc "if" Formula cond "then" Formula then "else" Formula else
26   | non-assoc "if" Formula cond "then" Formula
27   > "forall" {Decl ","}* "|" Formula
28   | "exists" {Decl ","}* "|" Formula;
29
30 syntax Decl = {Id ",",}* vars ":" Expr expr;
31
32 syntax Expr
33   = brackets: "(" Expr ")"
34   > var: Id
35   | "|" Expr "|"
36   > fieldAccess: Expr "." Id
37   | trans: Expr "." "^" Id
38   | reflTrans: Expr "." "*" Id
39   | functionCall: Id func "(" {Expr ","}* actuals ")"
40   | instanceAccess: Expr spc "[" Id inst"]"
41   | Lit
42   > nextVal: Expr "\'"
43   > "-" Expr
44   > left Expr lhs "*" Expr rhs
45   | non-assoc Expr lhs "/" Expr rhs
46   | non-assoc Expr lhs "%" Expr rhs
47   > left Expr lhs "+" Expr rhs
48   | non-assoc Expr lhs "-" Expr rhs
49   | left Expr lhs "++" Expr rhs
50   > "{" Decl d "|" Formula form "}";
51
52 syntax Lit
53   = Int
54   | StringConstant
55   | setLit: "{" {Expr ","}* elems "}";
56   | "none";
57
58 syntax Type
59   = TypeName tp
60   | "set" TypeName tp
61   | "?" TypeName tp;

```

Listing B.6: Rascal definition of common REBEL2 syntax.

```

1  syntax Part
2    = Config cfg
3    | Assert asrt
4    | Check chk
5    ;
6
7  syntax Config = "config" Id name "=" {InstanceSetup ","}+ instances ";" ;
8
9  syntax InstanceSetup
10   = {Id ","}+ labels ":" Type spec Mocks? mocks Forget?
11     forget InState? inState WithAssignments? assignments
12     | Id label WithAssignments assignments;
13
14  syntax Mocks = "mocks" Type concrete;
15  syntax Forget = "forget" {Id ","}+ fields;
16  syntax InState = "is" State state;
17  syntax WithAssignments = "with" {Assignment ","}+ assignments;
18  syntax Assignment = Id fieldName "=" Expr val;
19  syntax Assert = "assert" Id name "=" Formula form ";" ;
20
21  syntax Formula
22   = non-assoc "if" Formula cond "then" Formula then "else" Formula else
23     > TransEvent event "on" Expr var WithAssignments? with
24     > "next" Formula form
25     | "first" Formula form
26     | "last" Formula form
27     > "eventually" Formula form
28     | "always" Formula form
29     | "always-last" Formula form
30     | right Formula first "until" Formula second
31     | right Formula first "release" Formula second;
32
33  syntax TransEvent = wildcard: "*";
34
35  syntax Check
36   = Command cmd Id name "from" Id config "in" SearchDepth depth Objectives? objs Expect? expect";";
37
38  syntax Command
39   = "check"
40     | "run";
41
42  syntax SearchDepth
43   = "max" Int steps "steps"
44     | "exact" Int steps "steps";
45
46  syntax Objectives
47   = "with" {Objective ","}+ objs;
48
49  syntax Objective
50   = "minimal" Expr expr
51     | "maximal" Expr expr
52     | "infinite" "trace"
53     | "finite" "trace";
54
55  syntax Expect
56   = "expect" ExpectResult;
57
58  syntax ExpectResult
59   = "trace"
60     | "no" "trace";
61

```

Listing B.7: Rascal definition of REBEL2 'check' syntax.

```

1  syntax Part = Spec spc;
2
3  syntax Spec = "spec" Id name Instances? instances Fields? fields
4      Event* events Pred* preds Fact* facts States? states;
5
6  syntax Instances = "[" {Instance ","}* instances "]" ;
7
8  syntax Instance
9      = Id
10     | Id "*";
11
12 syntax Fields = {Field ","}* fields ";";
13 syntax Field = Id name ":" Type tipe;
14
15 syntax Event = Modifier* modifiers "event" Id name "(" {FormalParam ","}* params ")" EventBody body;
16 syntax Modifier = "init" | "final" | "internal";
17 syntax FormalParam = Id name ":" Type tipe;
18
19 syntax EventBody = Pre? pre Post? post EventVariant* variants;
20 syntax Pre = "pre" ":" {Formula ","}* formulas ";";
21 syntax Post = "post" ":" {Formula ","}* formulas ";";
22 syntax EventVariant = "variant" Id name EventVariantBody body;
23 syntax EventVariantBody = Pre? pre Post? post;
24 syntax Pred = "pred" Id name "(" {FormalParam ","}* params ")" "=" Formula form ";";
25
26 syntax Fact = "assume" Id name "=" Formula form ";";
27
28 syntax States = "states" ":" StateBlock root;
29 syntax StateBlock = InnerStates? inner Transition* trans;
30
31 syntax Transition
32     = State from "->" State to ":" {TransEvent ","}* events ";";
33     | Id super "{" StateBlock child "}";
34
35 syntax InnerStates = "[" {Id ","}* states "]" ;
36
37 syntax State = QualifiedName name | "(*)";
38
39 syntax TransEvent
40     = QualifiedName event \ "empty"
41     | "empty";
42
43 syntax Lit = "this";

```

Listing B.8: Rascal definition of REBEL2 ‘Specification’ syntax.

ALGORITHMS

C.1 ALLEALE ALGORITHMS

addDistinct The function `ADD DISTINCT` adds a tuple to a relation, ensuring that it will be distinct from other tuples in the relation by adding constraints to the `ATTCONS` column:

```

1: function ADDDISTINCT( $r$ : REL,  $t$ : TUPLE)
2:   for  $t' \leftarrow r_{body}$ ,  $t \neq t'$ , canOverlap( $t, t'$ ) do
3:      $t_{attCons} \leftarrow t_{attCons} \wedge (\neg t'_{exists} \vee \neg attEqual(t, t'))$ 
4:   end for
5:    $r_{body} \leftarrow r_{body} \cup \{t\}$ 
6:   return  $r$ 
7: end function

```

canOverlap The function `CANOVERLAP` returns \top when two tuples are indistinguishable with respect to their attribute values:

```

1: function CANOVERLAP( $t$ : TUPLE,  $t'$ : TUPLE)
2:   for  $a \leftarrow t_{attributes}$ ,  $a' \leftarrow t'_{attributes}$ ,  $a_{name} = a'_{name}$  do
3:     if  $a_{value} \neq ? \wedge a'_{value} \neq ? \wedge a_{value} \neq a'_{value}$  then
4:       return  $\perp$  ▷ non-'holes', different values, overlap impossible
5:     end if
6:   end for
7:   return  $\top$ 
8: end function

```

attEqual The function `ATTEQUAL` constructs a constraint to ensure that two tuples will be equal.

```

1: function ATTEQUAL( $t$ : TUPLE,  $t'$ : TUPLE)
2:    $\delta \leftarrow \top$ 
3:   for  $a \leftarrow t_{attributes}$ ,  $a' \leftarrow t'_{attributes}$ ,  $a_{name} = a'_{name}$  do
4:      $\delta \leftarrow \delta \wedge a_{value} = a'_{value}$  ▷ force atts to have same values
5:   end for
6:   return  $\delta$ 
7: end function

```

C.2 KEY ALGORITHMS FROM THE REBEL2 IMPLEMENTATION

The coming sections contain the key algorithms of the REBEL2 language, namely the implementation of the **forget** and **mock** operators. We implemented these algorithms in Rascal, which is what is shown in the listings in this appendix. See Appendix A.1 for a brief overview of Rascal.

C.2.1 *The 'Forget' algorithm*

Listing C.1 shows the Rascal implementation of the **forget** algorithm. The essence of this algorithm is that it removes all references to the fields that need to be 'forgotten' from the specifications to check. The algorithm operates on the Concrete Syntax Tree (CST) of the parsed and type checked specifications and performs a source-to-source transformation where the target is still in the same Rebel2 syntax. Formula's which have a term which references a forgotten field are removed as a whole from the **pre**(conditions), **post**(conditions) and facts. Removing the entire formula on a single term that must be forgotten may not be strictly necessary: the term could also be reduced to true. This could however unintentionally change the meaning of the whole formula. The ripple effect of such an operation might be hard to foresee by users thus we chose to the easier to follow approach of removing the whole formula.

C.2.2 *The 'Mock' algorithm*

Listing C.4 contains the **mock** algorithm. Like the **forget** algorithm, the **mock** algorithm performs a source-to-source transformation on the language level of REBEL2. The algorithm performs a replace operation by renaming all type references of the mock specification with the name of the original specification. It is a similar operation as is performed by a renaming refactoring in a statically typed language such as Java. It performs this renaming operations in the context of the *mock* specifications. The reason is that the change is then local to the mocked specification. In the end, the name of the mocked specification is replaced by the name of the original specification (see line 12 of Listing C.4).

```

1 // RebelDependency = <Module m, TModel tm, datetime timeOfParse> where
2 // Module = The root node of the concrete syntax tree (CST) of a parsed Rebel2 text file.
3 // TModel = TypeModel containing gathered type information
4 // (see https://www.rascal-mpl.org/docs/Packages/typepal/TypePal/)
5
6 // modDep = Graph[RebelDependency], a dependency graph of type checked Rebel2 specifications
7 // that import each other
8
9 // Extract a graph only containing the Rebel2 Specifications without the attached TypeModels
10 Graph[Spec] spcDep = extractSpecDependencyGraph(modDep);
11
12 // Merge all use-def relations from the type models (tm).
13 // * = the Rascal 'splat' operator for flattening all elements in underlying containers
14 rel[loc,loc] globDefUse = {*dep1.tm.useDef<1,0>,*dep2.tm.useDef<1,0> |
15 <RebelDependency dep1, RebelDependency dep2> <- modDep};
16
17 bool slice(set[Id] fields) {
18   set[Spec] allSpecs = {s | Spec s <- spcDep<0>+spcDep<1>};
19
20   // Loop over all fields to be 'forgotten'
21   for (Id field <- fields) {
22     Field fld = lookupFieldByRef(tm.useDef[field@loc], spcDep);
23
24     // Find all the uses of the field in all included specifications.
25     set[loc] uses = globDefUse[fld.name@loc];
26
27     // Remove all fields, formula's in pre and post and
28     // facts that reference the 'forgotten' field
29     allSpecs = visit(allSpecs) {
30       case Spec s => filterFieldAndFacts(fld, s, uses)
31       case Pre pre => filterPre(pre, uses)
32       case Post post => filterPost(post, uses)
33     }
34
35     // Remove all parameters that became unused
36     allSpecs = visit(allSpecs) {
37       case Event e => filterParameters(e, globDefUse)
38     }
39   }
40
41   // Replace all occurrences of the specs in the spec dependency graph
42   spcDep = visit(spcDep) {
43     case Spec orig => changed
44     when Spec changed <- allSpecs,
45         changed@loc == orig@loc
46   }
47
48   return true;
49 }

```

Listing C.1: The Rascal implementation of the 'Forget' (slice) algorithm. The algorithm removes all references to fields that are to be 'forgotten'. It is a source-to-source transformation on the language level of REBEL2.

```

1 private Spec filterFieldAndFacts(Field fld, Spec s, set[loc] uses)
2   = filterFacts(filterField(s, fld), uses);
3
4 private Spec filterField(spc:(Spec)'spec <Id name> <Instances? inst> <Fields? flds>
5   <Constraints? cons> <Event* evnts> <Pred* preds> <Fact* fcts> <States? sts>', Field fld) {
6
7   if (size({f | /Field f := flds}) == 1, /fld := flds) {
8     return ((Spec)'spec <Id name> <Instances? inst> <Constraints? cons>
9       <Event* evnts> <Fact* fcts> <States? sts>')[@loc=spc@loc];
10  }
11
12  Fields filterFields((Fields)'<Field f>, <{Field ", "}>+ after;')
13    = (Fields)'<Field ", "}>+ after; ' when f == fld;
14
15  Fields filterFields((Fields)'<Field ", "}>+ before, <Field f>;')
16    = (Fields)'<Field ", "}>+ before; ' when f == fld;
17
18  Fields filterFields((Fields)'<Field ", "}>+ before, <Field f>, <{Field ", "}>+ after;')
19    = (Fields)'<Field ", "}>+ before, <{Field ", "}>+ after; ' when f == fld;
20
21  return visit(spc) {
22    case Fields ff => filterFields(ff) when /fld := ff
23  }
24 }
25
26 private Spec filterFacts(Spec spc, set[loc] uses) {
27   Spec filterFact(s:(Spec)'spec <Id name> <Instances? inst> <Fields? flds>
28     <Constraints? cons> <Event* evnts> <Fact ff> <States? sts>', Fact f)
29     = (Spec)'spec <Id name> <Instances? inst> <Fields? flds>
30       <Constraints? cons> <Event* evnts> <States? sts>')[@loc=s@loc]
31       when ff == f;
32
33   Spec filterFact(s:(Spec)'spec <Id name> <Instances? inst> <Fields? flds>
34     <Constraints? cons> <Event* evnts> <Fact ff> <Fact* other> <States? sts>', Fact f)
35     = (Spec)'spec <Id name> <Instances? inst> <Fields? flds>
36       <Constraints? cons> <Event* evnts> <Fact* other> <States? sts>')[@loc=s@loc]
37       when ff == f;
38
39   Spec filterFact(s:(Spec)'spec <Id name> <Instances? inst> <Fields? flds>
40     <Constraints? cons> <Event* evnts> <Fact* other> <Fact ff> <States? sts>', Fact f)
41     = (Spec)'spec <Id name> <Instances? inst> <Fields? flds>
42       <Constraints? cons> <Event* evnts> <Fact* other> <States? sts>')[@loc=s@loc]
43       when ff == f;
44
45   Spec filterFact(s:(Spec)'spec <Id name> <Instances? inst> <Fields? flds>
46     <Constraints? cons> <Event* evnts> <Fact* before> <Fact ff> <Fact* after> <States? sts>', Fact f)
47     = (Spec)'spec <Id name> <Instances? inst> <Fields? flds>
48       <Constraints? cons> <Event* evnts> <Fact* before> <Fact* after> <States? sts>')[@loc=s@loc]
49       when ff == f;
50
51   // Remove all the formulas from the facts in which a field that is to be 'forgotten' is referenced
52   for (loc use <- uses, Fact f <- spc.facts, isContainedIn(use, f@loc)) {
53     spc = filterFact(spc, f);
54   }
55
56   return spc;
57 }

```

Listing C.2: The Rascal implementation of the functions that remove the field definition from the specs (the `filterFields` function) and the formulas in the facts that reference these fields (the `filterFacts` function).

```

1 private Pre filterPre(Pre pre, set[loc] uses) {
2   Pre filterPre(p:(Pre)'pre: <Formula ff>;',
3     Formula f)
4     = (Pre)'pre: ;'[@\loc=p@\loc] when ff == f;
5
6   Pre filterPre(p:(Pre)'pre: <{Formula ", "}* form>, <Formula ff>;',
7     Formula f)
8     = (Pre)'pre: <{Formula ", "}* form>;'[@\loc=p@\loc] when ff == f;
9
10  Pre filterPre(p:(Pre)'pre: <Formula ff>, <{Formula ", "}* form>;',
11    Formula f)
12    = (Pre)'pre: <{Formula ", "}* form>;'[@\loc=p@\loc] when ff == f;
13
14  Pre filterPre(p:(Pre)'pre: <{Formula ", "}* before>, <Formula ff>, <{Formula ", "}* after>;',
15    Formula f)
16    = (Pre)'pre: <{Formula ", "}* before>,
17      <{Formula ", "}* after>;'[@\loc=p@\loc] when ff == f;
18
19  for (loc use <- uses,
20      isContainedIn(use, pre@\loc),
21      Formula f <- pre.formulas,
22      isContainedIn(use, f@\loc)) {
23    pre = filterPre(pre, f);
24  }
25
26  return pre;
27 }
28
29 private Post filterPost(Post post, set[loc] uses) {
30   Post filterPost(p:(Post)'post: <Formula ff>;', Formula f)
31     = (Post)'post: ;'[@\loc=p@\loc] when ff == f;
32
33   Post filterPost(p:(Post)'post: <{Formula ", "}* form>, <Formula ff>;',
34     Formula f)
35     = (Post)'post: <{Formula ", "}* form>;'[@\loc=p@\loc] when ff == f;
36
37   Post filterPost(p:(Post)'post: <Formula ff>, <{Formula ", "}* form>;',
38     Formula f)
39     = (Post)'post: <{Formula ", "}* form>;'[@\loc=p@\loc] when ff == f;
40
41   Post filterPost(p:(Post)'post: <{Formula ", "}* before>, <Formula ff>, <{Formula ", "}* after>;',
42     Formula f)
43     = (Post)'post: <{Formula ", "}* before>,
44       <{Formula ", "}* after>;'[@\loc=p@\loc] when ff == f;
45
46   for (loc use <- uses,
47       isContainedIn(use, post@\loc),
48       Formula f <- post.formulas,
49       isContainedIn(use, f@\loc)) {
50     post = filterPost(post, f);
51   }
52
53   return post;
54 }

```

Listing C.3: The Rascal implementation of the functions that remove the formulas from the pre- and postconditions that reference these fields (the `filterFacts` function).

```

1 bool replace(Type abstractSpctype, Type concreteSpctype) {
2   Spec abstractSpctype = lookupSpecByRef(tm.useDef[abstractSpctype@\loc], deps);
3   Spec concreteSpctype = lookupSpecByRef(tm.useDef[concreteSpctype@\loc], deps);
4
5   Spec unalteredAbstractSpctype = abstractSpctype;
6
7   abstractSpctype = visit(abstractSpctype) {
8     case Type t => concreteSpctype when "<t>" == "<abstractSpctype>"
9     case Expr e => [Expr]"<concreteSpctype>" when "<e>" == "<abstractSpctype>"
10  };
11
12  abstractSpctype.name = concreteSpctype.name;
13
14  spcDep += {<f,abstractSpctype> | <f,t> <- spcDep, t == concreteSpctype};
15  spcDep = {<f,t> | <Spec f, Spec t> <- spcDep, f != concreteSpctype, t != concreteSpctype};
16
17  // remove the original mock spec from the dependencies
18  spcDep -= {<unalteredAbstractSpctype,unalteredAbstractSpctype>};
19
20  return true;
21 }
22
23 // Config cfg = the referenced configuration in the check statement to be executed
24
25 cfg = visit (cfg) {
26   case (InstanceSetup)'<{Id ","}+ labels> : <Type abstractSpctype> mocks <Type concreteSpctype>
27     <Forget? forget> <InState? inState> <WithAssignments? assignments>'
28     =>
29     (InstanceSetup)'<{Id ","}+ labels> : <Type concreteSpctype> <Forget? forget> <InState? inState>
30     <WithAssignments? assignments>' when replace(abstractSpctype, concreteSpctype)
31 };

```

Listing C.4: The Rascal implementation of the 'mock' (replace) algorithm.

C.3 APPLYING 'FORGET' AND 'MOCK'

Listing C.5 contains the code that applies both the **forget** and **mock** algorithms. Both algorithms are run inside the same closure manipulating the shared variable `spcDep`, which holds the CST's of the parsed specifications. This altered `spcDep` is then type checked again (Listing C.5, line 21) to see if the resulting specification is correct. In the case that a user applied a mock that is not interface compatible, the type checker will report this error.

```
1 // Step 1: Apply Mock
2 cfg = visit (cfg) {
3   case (InstanceSetup)'<{Id ","}+ labels> : <Type abstractSpc> mocks <Type concreteSpc>
4     <Forget? forget> <InState? inState> <WithAssignments? assignments>' =>
5     (InstanceSetup)'<{Id ","}+ labels> : <Type concreteSpc> <Forget? forget>
6     <InState? inState> <WithAssignments? assignments>'
7     when replace(abstractSpc, concreteSpc)
8 };
9
10 // Step 2: Apply Forget (removing 'forgotten' fields)
11 cfg = visit (cfg) {
12   case (InstanceSetup)'<{Id ","}+ labels> : <Type spc> forget <{Id ","}+ fields>
13     <InState? inState> <WithAssignments? assignments>' =>
14     (InstanceSetup)'<{Id ","}+ labels> : <Type spc> <InState? inState> <WithAssignments? assignments>'
15     when slice({f | f <- fields})
16 };
17
18 set[Spec] filteredSpecs = filterNonReferencedSpecs(spcDep, tm, cfg);
19
20 Module gen = assembleModule(root.\module.name, filteredSpecs, as, cfg, chk);
21 TModel genTm = rebelTModelFromModule(gen, {}, pcfg);
```

Listing C.5: Applying the forget and mock algorithms.

C.4 TRANSLATION FROM REBEL2 TO ALLEALLE

This section contains the listings of the translation functions translating a REBEL2 specification to a ALLEALLE specification. The translation is split in several parts. Listing C.6 contains the overall algorithm.

```
1  alias TransResult = tuple[str alleSpec, int duration];
2
3  TransResult translateToAlleAlle(Config cfg, Module m, TModel tm, PathConfig pcfg,
4  bool saveAlleAlleFile = true) {
5  RelMapping rm = constructRelMapping(m, tm);
6
7  set[Spec] spcs = {inst.spc | inst <- cfg.instances};
8
9  str alleSpec = "<translateRelationDefinitions(cfg, tm)>
10             '<translateConstraints(cfg, spcs, tm, rm)>
11             '<translateFacts(m, rm, tm, spcs)>
12             '<translateAssert(m, rm, tm, spcs)>
13             '
14             '// Minimize the number of steps by minimizing the number of Configurations
15             'objectives: minimize Config[count]";
16
17
18  if (saveAlleAlleFile) {
19    writeFile(addModuleToBase(pcfg.checks, m)[extension="alle"], alleSpec);
20  }
21
22  return <alleSpec, duration>;
23 }
24
25 str translateFacts(Module m, RelMapping rm, TModel tm, set[Spec] spcs) {
26 int lastUnique = 0;
27 int nxtUnique() { return lastUnique += 1;}
28 Context ctx = ctx(rm, tm, spcs, true, defaultCurRel(), defaultStepRel(), nxtUnique);
29
30 return "<for (/Spec s <- m.parts) {>// Facts from spec '<s.name>'
31       '<for (Fact f <- s.facts) {>// Fact '<f.name>'
32       '<translate(f.form, ctx)>
33       '<>><>>";
34 }
35
36 str translateAssert(Module m, RelMapping rm, TModel tm, set[Spec] spcs) {
37 set[Assert] asserts = {as | /Assert as <- m.parts};
38 if (size(asserts) > 1) {
39   throw "There should be only one assert to translate";
40 }
41
42 int lastUnique = 0;
43 int nxtUnique() { lastUnique += 1; return lastUnique; }
44 Context ctx = ctx(rm, tm, spcs, true, defaultCurRel(), defaultStepRel(), nxtUnique);
45
46 return "<for (Assert a <- asserts) {>// Assert '<a.name>'
47       '<translate(a.form, ctx)><>>";
48 }
```

Listing C.6: The Rascal implementation of translating REBEL2 to ALLEALLE.

As a first step the normalized REBEL2 specification is annotated with ALLEALLE relations (Listing C.6, line 5). Listing C.7 contains this complete annotation algorithm.

The result is a map mapping a location in the REBEL2 specification to a tuple consisting of the ALLEALLE relational expression and the relational header.

```

1  alias RelMapping = map[loc, RelExpr];
2  alias RelExpr = tuple[str relExpr, Heading heading];
3  alias Heading = map[str, Domain];
4
5  data Domain
6    = idDom()
7    | intDom()
8    | strDom()
9    ;
10
11 data AnalysisContext = actx(RelExpr (loc l) lookup, void (loc l, RelExpr r) add, str curRel,
12   str stepRel, TModel tm, map[loc, Spec] specs, set[str] emptySpecs,
13   void (loc, str) addCurRelScoped, str (loc) lookupCurRelScoped);
14
15 AnalysisContext nextCurRel(AnalysisContext old) = actx(old.lookup, old.add, getNextCurRel(old.curRel),
16   old.stepRel, old.tm, old.specs, old.emptySpecs, old.addCurRelScoped, old.lookupCurRelScoped);
17
18 AnalysisContext newCurRel(str newCurRel, AnalysisContext old) = actx(old.lookup, old.add, newCurRel,
19   old.stepRel, old.tm, old.specs, old.emptySpecs, old.addCurRelScoped, old.lookupCurRelScoped);
20
21 AnalysisContext nextStepRel(AnalysisContext old) = actx(old.lookup, old.add, old.curRel,
22   getNextStepRel(old.stepRel), old.tm, old.specs, old.emptySpecs,
23   old.addCurRelScoped, old.lookupCurRelScoped);
24
25 AnalysisContext nextCurAndStepRel(AnalysisContext old) = actx(old.lookup, old.add,
26   getNextCurRel(old.curRel), getNextStepRel(old.stepRel), old.tm, old.specs,
27   old.emptySpecs, old.addCurRelScoped, old.lookupCurRelScoped);
28
29 RelMapping constructRelMapping(Module m, TModel tm) {
30   RelMapping mapping = ();
31   void addRel(loc l, RelExpr r) { mapping[l] = r; }
32   RelExpr lookupRel(loc l) = mapping[l] when l in mapping;
33   default RelExpr lookupRel(loc l) { throw "No Relation expression stored for location '<l>'; }
34
35   map[loc, str] curRelScoped = ();
36   void addCurRelScoped(loc l, str r) { curRelScoped[l] = r; }
37   str lookupCurRelScoped(loc l) = curRelScoped[l] when l in curRelScoped;
38   default str lookupCurRelScoped(loc l) { throw "No current relation stored for expression at <l>"; }
39
40   map[loc, Spec] specs = (s@\loc : s | /Spec s <- m.parts);
41   set[str] emptySpecs = findEmptySpecs(m);
42
43   AnalysisContext ctx = actx(lookupRel, addRel, defaultCurRel(), defaultStepRel(), tm, specs,
44     emptySpecs, addCurRelScoped, lookupCurRelScoped);
45
46   // First do all the events in the specification
47   for (/Spec s <- m.parts) {
48     for (Event ev <- s.events) {
49       analyse(ev, "<s.name>", ctx);
50     }
51
52     for (Fact f <- s.facts) {
53       analyse(f, ctx);
54     }
55   }
56
57   for (/Assert a <- m.parts) {
58     analyse(a, ctx);
59   }
60

```

```

61     return mapping;
62 }
63
64 private set[str] findEmptySpecs(Module m) = {"<s.name>" | /Spec s <- m.parts, isEmptySpec(s)};
65 private bool isEmptySpec(Spec spc) = /Transition _ != spc.states;
66
67 void analyse(current:(Event)'<Modifier* _> event <Id name>(<{FormalParam " ," }* params>)
68 <EventBody body>', str specName, AnalysisContext ctx) {
69     // Add relations for parameters
70     for (FormalParam p <- params) {
71
72         str fldName = isPrim(p.name@loc, ctx) ? "<p.name>" : "instance";
73         ctx.add(p.name@loc, "<p.name>", (fldName : type2Dom(getType(p.name@loc, ctx))));
74     }
75
76     analyse(body, ctx);
77 }
78
79 void analyse((EventBody)'<Pre? maybePre> <Post? maybePost> <EventVariant* variants>',
80 AnalysisContext ctx) {
81     for (/Pre pre := maybePre, Formula f <- pre.formulas) {
82         analyse(f, ctx);
83     }
84
85     for (/Post post := maybePost, Formula f <- post.formulas) {
86         analyse(f, ctx);
87     }
88
89     // There should not be any variants any more since the analyzer should run on
90     // normalized specifications only
91     if (/EventVariant v := variants) {
92         throw "Can not analyse events with variants. Analyzer only handles normalized specifications";
93     }
94 }
95
96 void analyse(Fact f, AnalysisContext ctx) = analyse(f.form, ctx);
97 void analyse(Assert a, AnalysisContext ctx) = analyse(a.form, ctx);
98
99 // From Common Syntax
100 void analyse((Formula)'<Formula f>', AnalysisContext ctx) = analyse(f, ctx);
101 void analyse((Formula)'!<Formula f>', AnalysisContext ctx) = analyse(f, ctx);
102 void analyse((Formula)'<Expr spc>.<Id event>(<{Expr " ," }* actuals>)', AnalysisContext ctx) {
103     analyse(spc, ctx);
104
105     for (Expr arg <- actuals) {
106         analyse(arg, ctx);
107     }
108 }
109
110 void analyse((Formula)'<Expr spc> is <QualifiedName state>', AnalysisContext ctx) = analyse(spc, ctx);
111 void analyse((Formula)'<Expr lhs> in <Expr rhs>', AnalysisContext ctx)
112 { analyse(lhs, ctx); analyse(rhs, ctx); }
113 void analyse((Formula)'<Expr lhs> notin <Expr rhs>', AnalysisContext ctx)
114 { analyse(lhs, ctx); analyse(rhs, ctx); }
115 void analyse((Formula)'<Expr lhs> \< <Expr rhs>', AnalysisContext ctx)
116 { analyse(lhs, ctx); analyse(rhs, ctx); }
117 void analyse((Formula)'<Expr lhs> \<= <Expr rhs>', AnalysisContext ctx)
118 { analyse(lhs, ctx); analyse(rhs, ctx); }
119 void analyse((Formula)'<Expr lhs> = <Expr rhs>', AnalysisContext ctx)
120 { analyse(lhs, ctx); analyse(rhs, ctx); }
121 void analyse((Formula)'<Expr lhs> != <Expr rhs>', AnalysisContext ctx)
122 { analyse(lhs, ctx); analyse(rhs, ctx); }
123 void analyse((Formula)'<Expr lhs> \>= <Expr rhs>', AnalysisContext ctx)
124 { analyse(lhs, ctx); analyse(rhs, ctx); }

```

```

125 void analyse((Formula)'<Expr lhs> \> <Expr rhs>', AnalysisContext ctx)
126 { analyse(lhs, ctx); analyse(rhs,ctx); }
127 void analyse((Formula)'<Formula lhs> && <Formula rhs>', AnalysisContext ctx)
128 { analyse(lhs, ctx); analyse(rhs,ctx); }
129 void analyse((Formula)'<Formula lhs> || <Formula rhs>', AnalysisContext ctx)
130 { analyse(lhs, ctx); analyse(rhs,ctx); }
131 void analyse((Formula)'<Formula lhs> => <Formula rhs>', AnalysisContext ctx)
132 { analyse(lhs, ctx); analyse(rhs,ctx); }
133 void analyse((Formula)'<Formula lhs> \<=> <Formula rhs>', AnalysisContext ctx)
134 { analyse(lhs, ctx); analyse(rhs,ctx); }
135 void analyse((Formula)'if <Formula cond> then <Formula then> else <Formula els>', AnalysisContext ctx)
136 { analyse(cond, ctx); analyse(\then,ctx); analyse(els,ctx);}
137 void analyse((Formula)'if <Formula cond> then <Formula then>', AnalysisContext ctx)
138 { analyse(cond, ctx); analyse(\then,ctx);}
139
140 void analyse((Formula)'forall <Decl ", "+ decls> | <Formula f>', AnalysisContext ctx) {
141     for (Decl d <- decls) {
142         analyse(d,ctx);
143     }
144     analyse(f,ctx);
145 }
146
147 void analyse((Formula)'exists <Decl ", "+ decls> | <Formula f>', AnalysisContext ctx) {
148     for (Decl d <- decls) {
149         analyse(d,ctx);
150     }
151     analyse(f,ctx);
152 }
153
154 void analyse((Formula)'noOp(<Expr expr>)', AnalysisContext ctx) { analyse(expr,ctx); }
155
156 void analyse((Formula)'eventually <Formula f>', AnalysisContext ctx) = analyse(f,ctx);
157 void analyse((Formula)'always <Formula f>', AnalysisContext ctx) = analyse(f,ctx);
158 void analyse((Formula)'always-last <Formula f>', AnalysisContext ctx) = analyse(f,ctx);
159 void analyse((Formula)'next <Formula f>', AnalysisContext ctx) = analyse(f,ctx);
160 void analyse((Formula)'first <Formula f>', AnalysisContext ctx) = analyse(f,ctx);
161 void analyse((Formula)'last <Formula f>', AnalysisContext ctx) = analyse(f,ctx);
162
163 void analyse((Formula)'<Formula u> until <Formula r>', AnalysisContext ctx) {
164     analyse(u,ctx);
165     analyse(r,ctx);
166 }
167
168 void analyse((Formula)'<TransEvent event> on <Expr var> <WithAssignments? w>', AnalysisContext ctx) {
169     analyse(var,ctx);
170
171     for ((Assignment)'<Id fieldName> = <Expr val>' <- w) {
172         analyse(val,ctx);
173     }
174 }
175
176 void analyse((WithAssignments)'with <{Assignment ", "+ assignments}>', AnalysisContext ctx) {}
177
178 // From Common Syntax
179 void analyse(current:(Expr)'(<Expr expr>)', AnalysisContext ctx) {
180     analyse(expr,ctx);
181     ctx.add(current@\loc, ctx.lookup(expr@\loc));
182 }
183
184 void analyse(current:(Expr)'<Id var>', AnalysisContext ctx) {
185     analyse(var,ctx);
186     ctx.add(current@\loc, ctx.lookup(var@\loc));
187 }
188

```

```

189 void analyse(current:(Expr)'<Lit l>', AnalysisContext ctx) {
190     analyse(l,ctx);
191     ctx.add(current@\loc, ctx.lookup(l@\loc));
192 }
193
194 void analyse(current:(Id)'<Id var>', AnalysisContext ctx) {
195     Define def = getDefinition(var@\loc, ctx);
196
197     switch (def.idRole) {
198         case paramId(): ctx.add(current@\loc, ctx.lookup(def.defined));
199         case quantVarId(): ctx.add(current@\loc, "<var>", ctx.lookup(def.defined).heading);
200         case fieldId(): ctx.add(current@\loc,
201             "<(<getSpecName(current@\loc,ctx)><capitalize("<var>")> |x| <ctx.curRel>)",
202             ("instance":idDom()), "<var>": type2Dom(getType(current@\loc,ctx)))>);
203         case specId(): ctx.add(current@\loc,
204             "<(Instance |x| <capitalize("<var>")>)]instance]", ("instance":idDom()));
205         case specInstanceId(): ctx.add(current@\loc,
206             "<<getSpecName(current@\loc,ctx)><var>", ("instance":idDom()));
207         default: throw "Id expression at '<current@\loc>' with role '<def.idRole>'
208             'is used in a way not yet handled by the relation analyser";
209     }
210 }
211
212 void analyse(current:(Expr)'<Expr expr>.<Id fld>', AnalysisContext ctx) {
213     analyse(expr,ctx);
214     RelExpr exprRel = ctx.lookup(expr@\loc);
215
216     if (getType(expr@\loc, ctx) == stringType() && "<fld>" == "length") {
217         // built-in attribute on string
218         ctx.add(current@\loc, "<(<exprRel.relExpr>)[<getFieldName(exprRel)>]",
219             ("<getFieldName(exprRel)>":type2Dom(intType()))>);
220     } else {
221         analyse(fld,ctx);
222         RelExpr fieldRel = ctx.lookup(fld@\loc);
223
224         ctx.add(current@\loc,
225             "<(<exprRel.relExpr>
226             '<renameIfNeeded(getFieldName(exprRel),\"instance\")> |x| <fieldRel.relExpr>[<fld>]",
227             ("<fld>":type2Dom(getType(fld@\loc,ctx)))>);
228     }
229 }
230
231 void analyse(current:(Expr)'<Expr expr>.*<Id fld>', AnalysisContext ctx) {
232     analyse(expr,ctx);
233     analyse(fld,ctx);
234
235     RelExpr exprRel = ctx.lookup(expr@\loc);
236     RelExpr fieldRel = ctx.lookup(fld@\loc);
237
238     ctx.add(current@\loc,
239         "<(<exprRel.relExpr><renameIfNeeded(getFieldName(exprRel),\"instance\")> |x|
240         '*(<fieldRel.relExpr>)[instance,<fld>]][<fld>]",
241         ("<fld>": type2Dom(getType(fld@\loc,ctx)))>);
242 }
243
244 void analyse(current:(Expr)'<Expr expr>.^<Id fld>', AnalysisContext ctx) {
245     analyse(expr,ctx);
246     analyse(fld,ctx);
247
248     RelExpr exprRel = ctx.lookup(expr@\loc);
249     RelExpr fieldRel = ctx.lookup(fld@\loc);
250
251     ctx.add(current@\loc, "<(<exprRel.relExpr><renameIfNeeded(getFieldName(exprRel),\"instance\")> |x|
252         '^(<fieldRel.relExpr>)[instance,<fld>]][<fld>]",

```

```

253         ("<fld>": type2Dom(getType(fld@\loc,ctx))));
254     }
255
256 void analyse(current:(Expr)'<Id func>(<{Expr ","}* actuals>)', AnalysisContext ctx) {
257     list[Expr] params = [p | p <- actuals];
258     for (p <- params) {
259         analyse(p,ctx);
260     }
261
262     switch ("<func>") {
263     case "substr": {
264         RelExpr strFld = ctx.lookup(params[0]@\loc);
265         ctx.add(current@\loc, "<strFld.relExpr>[<getFieldName(strFld)>]",
266             ("<getFieldName(strFld)>":type2Dom(stringType()))>);
267     }
268     case "toInt": {
269         RelExpr strFld = ctx.lookup(params[0]@\loc);
270         ctx.add(current@\loc, "<strFld.relExpr>[<getFieldName(strFld)>]",
271             ("<getFieldName(strFld)>":type2Dom(intType()))>);
272     }
273     case "toStr": {
274         RelExpr intFld = ctx.lookup(params[0]@\loc);
275         ctx.add(current@\loc, "<intFld.relExpr>[<getFieldName(intFld)>]",
276             ("<getFieldName(intFld)>":type2Dom(stringType()))>);
277     }
278     }
279 }
280
281 void analyse(current:(Expr)'<Expr spc>[<Id fld>]', AnalysisContext ctx) {
282     analyse(spc,ctx);
283     analyse(fld,ctx);
284
285     RelExpr exprRel = ctx.lookup(spc@\loc);
286     RelExpr fieldRel = ctx.lookup(fld@\loc);
287
288     ctx.add(current@\loc, "<fieldRel.relExpr>", ("instance":type2Dom(getType(fld@\loc,ctx))));
289 }
290
291 void analyse(current:(Expr)'<Expr expr>'', AnalysisContext ctx) {
292     analyse(expr, newCurRel("nxt", ctx));
293     ctx.add(current@\loc, ctx.lookup(expr@\loc));
294 }
295
296 void analyse(current:(Expr)'<Expr expr>', AnalysisContext ctx) {
297     analyse(expr, ctx);
298     ctx.add(current@\loc, ctx.lookup(expr@\loc));
299 }
300
301 void analyse(current:(Expr)'|<Expr expr>|', AnalysisContext ctx) {
302     analyse(expr, ctx);
303     AType tipe = getType(expr@\loc,ctx);
304     if (intType() := tipe) {
305         ctx.add(current@\loc, ctx.lookup(expr@\loc));
306     } else if (setType(_) := tipe || spectType(_) := tipe) {
307         RelExpr re = ctx.lookup(expr@\loc);
308         ctx.add(current@\loc, re);
309     }
310 }
311
312 private void setOperation(\loc current, Expr lhs, Expr rhs, str op, AnalysisContext ctx) {
313     RelExpr lhsRel = ctx.lookup(lhs@\loc);
314     RelExpr rhsRel = ctx.lookup(rhs@\loc);
315
316     str relExpr = "<lhsRel.relExpr> <op> <rhsRel.relExpr><renameIfNeeded(rhsRel, lhsRel)>";

```

```

317     ctx.add(current, <relExpr, lhsRel.heading>);
318 }
319
320 void analyse(current:(Expr)'<Expr lhs> + <Expr rhs>', AnalysisContext ctx) {
321     analyse(lhs,ctx);
322     analyse(rhs,ctx);
323
324     switch ({getType(lhs@loc, ctx), getType(rhs@loc, ctx)}) {
325         case {setType(AType tipe), tipe}: setOperation(current@loc, lhs, rhs, "+", ctx);
326         case {setType(AType tipe), setType(tipe)}: setOperation(current@loc, lhs, rhs, "+", ctx);
327         case {intType()}: ctx.add(current@loc, ctx.lookup(lhs@loc));
328     }
329 }
330
331 void analyse(current:(Expr)'<Expr lhs> - <Expr rhs>', AnalysisContext ctx) {
332     analyse(lhs,ctx);
333     analyse(rhs,ctx);
334
335     switch ({getType(lhs@loc,ctx), getType(rhs@loc,ctx)}) {
336         case {setType(AType tipe), tipe}: setOperation(current@loc, lhs, rhs, "- ", ctx);
337         case {setType(AType tipe), setType(tipe)}: setOperation(current@loc, lhs, rhs, "- ", ctx);
338         case {intType()}: ctx.add(current@loc, ctx.lookup(lhs@loc));
339     }
340 }
341
342 void analyse(current:(Expr)'<Expr lhs> * <Expr rhs>', AnalysisContext ctx) =
343     analyseBinOp(lhs,rhs,current@loc,ctx);
344 void analyse(current:(Expr)'<Expr lhs> / <Expr rhs>', AnalysisContext ctx) =
345     analyseBinOp(lhs,rhs,current@loc,ctx);
346 void analyse(current:(Expr)'<Expr lhs> % <Expr rhs>', AnalysisContext ctx) =
347     analyseBinOp(lhs,rhs,current@loc,ctx);
348
349 void analyseBinOp(Expr lhs, Expr rhs, loc complete, AnalysisContext ctx) {
350     analyse(lhs,ctx);
351     analyse(rhs,ctx);
352
353     ctx.add(complete, ctx.lookup(lhs@loc));
354 }
355
356 void analyse(current:(Expr)'<Expr lhs> ++ <Expr rhs>', AnalysisContext ctx) {
357     analyse(lhs,ctx);
358     analyse(rhs,ctx);
359
360     switch ({getType(lhs@loc, ctx), getType(rhs@loc, ctx)}) {
361         case {strType()}: ctx.add(current@loc, ctx.lookup(lhs@loc));
362     }
363 }
364
365 void analyse(current:(Expr)'{<Decl d> | <Formula f>}', AnalysisContext ctx) {
366     analyse(d,ctx);
367     analyse(f,ctx);
368
369     ctx.add(current@loc, ctx.lookup(d@loc));
370 }
371
372 void analyse(current:(Decl)'<{Id " ,"}+ vars>: <Expr expr>', AnalysisContext ctx) {
373     analyse(expr, ctx);
374
375     for (Id var <- vars) {
376         ctx.add(var@loc, ctx.lookup(expr@loc));
377         ctx.addCurRelScoped(var@loc, ctx.curRel);
378     }
379
380     ctx.add(current@loc, ctx.lookup(expr@loc));

```

```

381 }
382
383 void analyse(current:(Lit)'this', AnalysisContext ctx) {
384     ctx.add(current@loc, <"_<toLowerCase(getSpecName(current@loc,ctx))>", ("instance": idDom())>);
385 }
386
387 void analyse(current:(Lit)'<Int i>', AnalysisContext ctx) {
388     ctx.add(current@loc, <"_IntConst_<i>", ("const_<i>":intDom())>);
389 }
390
391 void analyse(current:(Lit)'none', AnalysisContext ctx) {
392     ctx.add(current@loc, <"_EMPTY", ("instance":idDom())>);
393 }
394
395 void analyse(current:(Lit)'<StringConstant s>', AnalysisContext ctx) {
396     ctx.add(current@loc, <"_StrConst_<unquote(s)>", ("const_<unquote(s)>":strDom())>);
397 }
398
399 void analyse(current:(Lit)'<{Expr ","}* elems>', AnalysisContext ctx) {
400     if ((Lit)'{' := current) {
401         ctx.add(current@loc, <"_EMPTY", ("instance":idDom())>);
402     }
403 }
404
405 private str unquote(StringConstant s) = "<s>"[1..-1];
406
407 private str getSpecName(loc l, AnalysisContext ctx) {
408     Define d = getDefinition(l, ctx);
409
410     if (d.defined in ctx.specs) {
411         return "<ctx.specs[d.defined].name>";
412     } else if (d.scope in ctx.specs) {
413         return "<ctx.specs[d.scope].name>";
414     } else {
415         println("Something is wrong: <d>");
416         throw "Unable to determine spec for '<d.id>' at <l>";
417     }
418 }
419
420 private Define getDefinition(loc use, AnalysisContext ctx) {
421     if ({loc def} := ctx.tm.useDef[use]) {
422         if (def in ctx.tm.definitions) {
423             return ctx.tm.definitions[def];
424         }
425     }
426
427     throw "Unable to define role";
428 }
429
430 private Maybe[Define] getDefinitionIfExists(loc use, AnalysisContext ctx) {
431     if (use notin ctx.tm.useDef<0>) {
432         return nothing();
433     }
434
435     if ({loc def} := ctx.tm.useDef[use]) {
436         if (def in ctx.tm.definitions) {
437             return just(ctx.tm.definitions[def]);
438         }
439     }
440
441     return nothing();
442 }
443
444

```

```

445 private str getFieldname(RelExpr re) {
446     if (size(re.heading) > 1) {
447         throw "More than 1 attribute in the relation, unable to determine field name";
448     }
449
450     return getOneFrom(re.heading);
451 }
452
453 private str renameIfNeeded(RelExpr lhs, RelExpr rhs) {
454     if (lhs.heading == rhs.heading) {
455         return "";
456     }
457
458     return "[<getFieldName(lhs)> as <getFieldName(rhs)>]";
459 }
460
461 private str renameIfNeeded(str lhs, str rhs) {
462     if (lhs == rhs) {
463         return "";
464     }
465
466     return "[<lhs> as <rhs>]";
467 }
468
469 private bool isPrim(loc expr, AnalysisContext ctx) = isPrim(t) when AType t := getType(expr, ctx);
470
471 private bool isPrim(intType()) = true;
472 private bool isPrim(stringType()) = true;
473 private default bool isPrim(AType t) = false;
474
475 private AType getType(loc expr, AnalysisContext ctx) = ctx.tm.facts[expr] when expr in ctx.tm.facts;
476 private default AType getType(loc expr, AnalysisContext ctx) {
477     throw "No type information known for expression at '<expr>'";
478 }
479
480 private Domain type2Dom(intType()) = intDom();
481 private Domain type2Dom(stringType()) = strDom();
482 private default Domain type2Dom(AType t) = idDom();
483
484 str dom2Str(intType()) = "int";
485 str dom2Str(stringType()) = "str";
486 default str dom2Str(AType t) = "id";
487
488 private str getNextCurRel(str oldCurRel) {
489     if (oldCurRel == defaultCurRel()) {
490         return "<defaultCurRel()>_1";
491     } else if (/cur_<n:[0-9]+/ := oldCurRel) {
492         return "<defaultCurRel()>.<toInt(n)+1>";
493     }
494
495     throw "Should not happen";
496 }
497
498 private str getNextStepRel(str oldStepRel) {
499     if (oldStepRel == defaultStepRel()) {
500         return "<defaultStepRel()>_1";
501     } else if (/step_<n:[0-9]+/ := oldStepRel) {
502         return "<defaultStepRel()>.<toInt(n)+1>";
503     }
504
505     throw "Should not happen";
506 }
507
508 private str defaultCurRel() = "cur";

```



```
509 private str defaultStepRel() = "step";
```

Listing C.7: The algorithm that annotates each expression in the REBEL2 CST with the to be constructed ALLEALLE relation.

The next part of the translation algorithm translates the normalized REBEL2 specification to an ALLEALLE specification, using the constructed relational mapping. Listing C.8 contains the relational translation algorithm.

```

1  str translateRelationDefinitions(Config cfg, TModel tm)
2  = "<translateStaticPart(cfg.instances<0>, tm)>
3  ,
4  '<translateDynamicPart(cfg, tm)>
5  ";
6
7  private str translateStaticPart(set[Spec] spcs, TModel tm) {
8  str def = "// Static configuration of state machines
9  '<buildSpecRel(spcs)>
10 '<buildStateRel(spcs,tm)>
11 '<buildAllowedTransitionRel(spcs)>
12 '<buildEventsAsSingleReIs(spcs)>
13 '<buildConstantReIs(spcs)>";
14
15 return def;
16 }
17
18 private str translateDynamicPart(Config cfg, TModel tm) {
19 str def = "// Dynamic configuration of state machines
20 '<buildConfigReIs(cfg.numberOfTransitions, cfg.finiteTrace, cfg.exactNrOfSteps)>
21 '<buildInstanceRel(cfg.instances)>
22 '<buildInstanceInStateRel(cfg.instances, cfg.numberOfTransitions, tm)>
23 '<buildRaisedEventsRel(cfg.instances<0,1>, cfg.numberOfTransitions, cfg.finiteTrace)>
24 '<buildChangedInstancesRel(cfg.instances<0,1>, cfg.numberOfTransitions, cfg.finiteTrace)>
25 '<buildStateVectors(lookupSpecs(cfg.instances), cfg, tm)>
26 '<buildEnumReIs(lookupSpecs(cfg.instances))>
27 '<buildEventParamReIs(lookupSpecs(cfg.instances), cfg, tm)>";
28
29 return def;
30 }
31
32 private str buildSpecRel(set[Spec] spcs)
33 = "// Define the specs that can take place in the transition system
34 '<for (s <- spcs) {}<buildSpecRel(s)>
35 '<>";
36
37 private str buildSpecRel(Spec spc)
38 = "<getCapitalizedSpecName(spc)> (spc:id) = {\<getLowerCaseSpecName(spc)>\}";
39
40 private str buildStateRel(set[Spec] spcs, TModel tm)
41 = "// Define all possible states for all machines
42 'State (state:id) = {\<stateTuplesWithDefaults>
43 'initialized (state:id) = {\<stateTuples>
44 'finalized (state:id) = {\<state_finalized>
45 'uninitialized (state:id) = {\<state_uninitialized>
46 '<buildIndividualStateReIs(spcs,tm)>"
47 when stateTuples := intercalate(",", [st | s <- spcs, str st := buildStateTuples(s,tm), st != ""]),
48 stateTuplesWithDefaults := intercalate(",", dup([st | s <- spcs,
49 str st := buildStateTuplesWithDefaultStates(s,tm), st != ""]));
50
51 private str buildIndividualStateReIs(set[Spec] spcs, TModel tm)
52 = "<for (s <- spcs) {}<buildIndividualStateRel(s,tm)>
53 '<>";

```

```

54
55 private str buildIndividualStateRel(Spec spc, TModel tm)
56 = "<for (str s <- states) {>State<getCapitalizedSpecName(spc)><capitalize(s)> (state:id) =
57   '{\<<getStateLabel(spc, s)>\>}'
58   '<>>'
59   when set[str] states := lookupStates(spc,tm);
60
61 private str buildStateTuples(Spec spc, TModel tm) = intercalate(", ", ["\<<s>\>" | str s <- states])
62   when set[str] states := lookupStateLabels(spc, tm);
63
64 private str buildStateTuplesWithDefaultStates(Spec spc, TModel tm) =
65   intercalate(", ", ["\<<s>\>" | str s <- states])
66   when set[str] states := lookupStateLabelsWithDefaultStates(spc, tm);
67
68 private str buildAllowedTransitionRel(set[Spec] spcs)
69 = "// Define which transitions are allowed
70   '//(in the form of 'from a state' -\> ' via an event' -\> 'to a state)'
71   'allowedTransitions (f:from:id, to:id, event:id) =
72     '{<intercalate(", ", [tt | s <- spcs, str tt := buildAllowedTransitionTuples(s), tt != ""]>)}';
73
74 private str buildAllowedTransitionTuples(Spec spc)
75 = intercalate(", ", ["\<<f><t><e>\>" | <f,t,e> <- flattenTransitions(spc)])
76   when /Transition _ := spc.states;
77
78 private default str buildAllowedTransitionTuples(Spec s) = "";
79
80 private str buildEventsAsSingleRels(set[Spec] spcs)
81 = "// Define each event as single relation so that the events can be used as variables
82   // in the constraints
83   '<for (r <- rels) {><r>'
84   '<>>'
85   when
86     set[str] rels := {buildSingleEventRel("<s.name>", e) | s <- spcs, /Event e := s.events};
87
88 private str buildSingleEventRel(str specName, Event e)
89 = "Event<capitalize(specName)><capitalize(event)> (event:id) =
90   '{\<event_<toLowerCase(specName)>_<toLowerCase(event)>\>}'
91   when str event := replaceAll("<e.name>", ":", "_");
92
93 private str buildConstantRels(set[Spec] spcs) {
94   set[str] constRels = {};
95
96   for ((Expr)'<Lit l>' := spcs) {
97     switch (l) {
98       case (Lit)'<Int i>': constRels += "___IntConst_<i> (const_<i>: int) = {\<<i>\>}'";
99       case (Lit)'{}': constRels += "___EMPTY (instance:id) = {}";
100      case (Lit)'none': constRels += "___EMPTY (instance:id) = {}";
101    }
102  }
103
104   return "<for (r <- constRels) {><r>
105     '<>>';
106 }
107
108 private str unquote(StringConstant s) = "<s>"[1..-1];
109
110 private rel[str,str,str] flattenTransitions(Spec s)
111 = {"<<cfrom>", "<cto>", "event_<name>_<event>"> |
112   str name := getLowerCaseSpecName(s),
113   //(Transition)'<State from> -\> <State to> : <{TransEvent " "}>+ events>;' := s.states,
114   str cfrom := convertFromState(from, name), str cto := convertToState(to, name),
115   str event <- {toLowerCase(replaceAll("<e>", ":", "_")) | TransEvent e <- events});
116
117 private str convertFromState((State)'(*)', str _) = "state_uninitialized";

```

```

118 private default str convertFromState(rebel::lang::SpecSyntax::State st, str spec) =
119   convertState(st, spec);
120
121 private str convertToState((State)'(*)', str _) = "state_finalized";
122 private default str convertToState(rebel::lang::SpecSyntax::State st, str spec) =
123   convertState(st, spec);
124
125 private str convertState(rebel::lang::SpecSyntax::State st, str spec) =
126   "state_<spec>_<toLowerCase(replaceAll("<st>", ":", "_"))>";
127
128 str translateConfigState(Spec spc, uninitialized()) = "state_uninitialized";
129 str translateConfigState(Spec spc, finalized()) = "state_finalized";
130 str translateConfigState(Spec spc, state(str name)) =
131   "state_<toLowerCase("<spc.name>")_<toLowerCase(replaceAll("<name>", ":", "_"))>";
132
133 private str buildEventParamRels(set[Spec] specs, Config cfg, TModel tm) {
134   list[str] rels = [];
135
136   for (Spec s <- specs, e <- lookupEvents(s), /FormalParam p <- e.params) {
137     rels += buildEventParamRel(s,e,p,cfg,tm);
138   }
139
140   return intercalate("\n", [r | r <- rels, r != ""]);
141 }
142
143 private str buildEventParamRel(Spec s, Event e, FormalParam p, Config cfg, TModel tm) {
144   list[str] defs = ["cur:id", "nxt:id", getParamHeaderDef(p,cfg)];
145   return "ParamEvent<getCapitalizedSpecName(s)><getCapitalizedEventName(e)>
146     '<getCapitalizedParamName(p)> (<intercalate(" ", defs)>) <buildParamTuples(s,e,p,cfg,tm)>";
147 }
148
149 private str buildParamTuples(Spec s, Event e, FormalParam p, Config cfg, TModel tm) {
150   void addTuple(int i, int j) {
151     if (isPrim(p.tipe, tm)) {
152       upperBound += "\<<i>,c<j>,?\>";
153     } else {
154       for (str otherInst <- getInstancesOfType(p.tipe, cfg.instances<0,l>, tm)) {
155         upperBound += "\<<i>,c<j>,<otherInst>\>";
156       }
157     }
158   }
159
160   list[str] upperBound = [];
161   for (int i <- [1..cfg.numberOfTransitions]) {
162     addTuple(i, i+1);
163   }
164
165   if (!cfg.finiteTrace) {
166     for (int i <- [cfg.numberOfTransitions..0]) {
167       addTuple(cfg.numberOfTransitions, i);
168     }
169   }
170
171   return "\<= {<intercalate(" ", upperBound)>}";
172 }
173
174 private str buildEnumRels(set[Spec] specs) {
175   list[str] rels = ["<s.name>_<instance> (instance:id) = {<<instance>\>}"];
176   Spec s <- specs, /Id instance <- s.instances;
177   return "<for (r <- rels) {<r>
178     '<>";
179 }
180
181 private str buildStateVectors(set[Spec] specs, Config cfg, TModel tm) {

```

```

182     list[str] rels = [buildFieldRel(s, f, cfg, tm) | Spec s <- specs, /Field f <- s.fields];
183     return "<for (r <- rels) {<r>
184         '<>';
185     }
186
187 private str buildFieldRel(Spec spc, Field f, Config cfg, TModel tm) {
188     list[str] defs = ["config:id", "instance:id", getFieldHeaderDef(f, cfg)];
189     return "<getCapitalizedSpecName(spc)><getCapitalizedFieldName(f)>
190         (<intercalate(" ", defs)>) <buildFieldTuples(spc, f, cfg, tm)>";
191 }
192
193 private str getFieldHeaderDef(Field f, Config cfg) = "<f.name><convertType(f.tipe)>";
194 private str getParamHeaderDef(FormalParam p, Config cfg) = "<p.name><convertType(p.tipe)>";
195
196 private str buildFieldTuples(Spec spc, Field f, Config cfg, TModel tm) {
197     list[str] lowerBound = [];
198     for (<str inst, "<f.name>", str val> <- cfg.initialValues[spc], val != "__none") {
199         lowerBound += "\<c1,<inst>,<val>\>";
200     }
201
202     list[str] upperBound = [];
203     for (str inst <- lookupInstances(spc, cfg.instances<0,1>), int i <- [1..cfg.numberOfTransitions+1]) {
204         if (!(i == 1 && /<inst, "<f.name>", "__none"> := cfg.initialValues[spc])) {
205             if (isPrim(f.tipe, tm)) {
206                 upperBound += "\<c<i>,<inst>,<?>\>";
207             } else if (isSetOfInt(f.tipe, tm)) {
208                 for (int j <- [1..cfg.maxSizeIntegerSets+1]) {
209                     upperBound += "\<c<i>,<inst>,<inst>_elem<j>\>";
210                 }
211             } else if (isSetOfString(f.tipe, tm)) {
212                 for (int j <- [1..cfg.maxSizeStringSets+1]) {
213                     upperBound += "\<c<i>,<inst>,<inst>_elem<j>\>";
214                 }
215             } else { // Set of other specification
216                 for (str otherInst <- getInstancesOfType(f.tipe, cfg.instances<0,1>, tm)) {
217                     upperBound += "\<c<i>,<inst>,<otherInst>\>";
218                 }
219             }
220         }
221     }
222
223     if (lowerBound != []) {
224         return "\>= {<intercalate(" ", lowerBound)>} \<= {<intercalate(" ", upperBound)>}";
225     } else {
226         return "\<= {<intercalate(" ", upperBound)>}";
227     }
228 }
229
230 private str buildChangedInstancesRel(rel[Spec, str] instances,
231     int numberOfTransitions, bool finiteTrace) {
232     list[str] tuples = ["\<c<c>,<c<j>,<i>\>" | int c <- [1..numberOfTransitions],
233         Spec s <- instances<0>, !isEmptySpec(s), str i <- instances[s]];
234
235     if (!finiteTrace) {
236         tuples += ["\<c<c>,<c<j>,<i>\>" | int c <- [numberOfTransitions..0], int j <- [c..0],
237             Spec s <- instances<0>, !isEmptySpec(s), str i <- instances[s]];
238     }
239
240     return "changedInstance (cur:id, nxt:id, instance:id) \<= {<intercalate(" ", tuples)>}
241         ";
242 }
243
244 private str buildRaisedEventsRel(rel[Spec spc, str instance] instances,
245     int numberOfTransitions, bool finiteTrace)

```

```

246 = "raisedEvent (cur:id, nxt:id, event:id, instance:id) \<=
247   {<intercalate(" ", [tups | <spc, i> <- instances,
248     str tups := buildRaisedEventsTuples(spc, i, numberOfTransitions, finiteTrace), tups != "" ]>}";
249
250 private str buildRaisedEventsTuples(Spec spc, str instance,
251   int numberOfTransitions, bool finiteTrace) {
252   list[str] tuples = ["\<<<<>,c<c+1>,<toLowerCase(event)>,<instance>\>" |
253     int c <- [1..numberOfTransitions], str event <- lookupRaisableEventName(spc)];
254
255   if (!finiteTrace) {
256     tuples += ["\<<<<>,c<j>,<toLowerCase(event)>,<instance>\>" |
257       int c <- [numberOfTransitions..0], int j <- [c..0], str event <- lookupRaisableEventName(spc)];
258   }
259
260   return intercalate(" ", tuples);
261 }
262
263 private str buildConfigRels(int numberOfTransitions, bool finiteTrace, bool exactNrOfSteps)
264 = "Config (config:id) \>= {\<c1>} \<=
265   '{<intercalate(" ", ["\<<<i>\>" | int i <- [1..numberOfTransitions+1]])>}'
266   '<if (exactNrOfSteps) {>order (cur:id, nxt:id) =
267     '{<intercalate(" ", ["\<<<i>,c<i+1>\>" | int i <- [1..numberOfTransitions]])>}<}' else {>
268     'order (cur:id, nxt:id) \<=
269     '{<intercalate(" ", ["\<<<i>,c<i+1>\>" | int i <- [1..numberOfTransitions]])>}<}'>
270   'first (config:id) = {\<c1>}
271   'last (config:id) \<= {\<intercalate(" ", ["\<<<i>\>" | int i <- [1..numberOfTransitions+1]])>}'
272   'back (config:id) <if (finiteTrace) {>=> {\<}' else {>
273     '\<= {\<intercalate(" ", ["\<<<i>\>" | int i <- [1..numberOfTransitions+1]])>}<}'>
274   'loop (cur:id, nxt:id) <if (finiteTrace) {>=> {\<}' else {>
275     '\<= {\<intercalate(" ", ["\<<<i>,c<j>\>" | int i <- [2..numberOfTransitions+1],
276       int j <- [1..i+1]])>}<}'>
277   ""';
278
279 private str buildInstanceRel(rel[Spec spc, str instance, State initialState] instances)
280 = "Instance (spec:id, instance:id) =
281   '{<intercalate(" ", ["\<<<toLowerCase("<inst.spc.name>")>,<inst.instance>\>" |
282     inst <- instances])>}";
283
284 private str buildInstanceInStateRel(rel[Spec spc, str instance, State state] instances,
285   int numberOfTransitions, TModel tm) {
286   str initialTup = buildInitialInstanceInStateTuples(instances,tm);
287   str iisRel = "instanceInState (config:id, instance:id, state:id) <if (initialTup != "") >
288     '\>=>{\<initialTup>}<}'>
289     '\<= {\<buildInstanceInStateTuples(instances<spc,instance>, numberOfTransitions, tm)>}'>";
290   return iisRel;
291 }
292
293 private str buildInitialInstanceInStateTuples(rel[Spec spc, str instance, State state] instances,
294   TModel tm)
295 = intercalate(" ", ["\<<<1,<i>,<translateConfigState(s, st)>\>" |
296   <s,i,st> <- instances, !isEmptySpec(s), st != noState()]);
297
298 private str buildInstanceInStateTuples(rel[Spec spc, str instance] instances,
299   int numberOfTransitions, TModel tm)
300 = intercalate(" ", ["\<<<<>,<i>,<toLowerCase(st)>\>" | int c <- [1..numberOfTransitions+1],
301   <s,i> <- instances, str st <- lookupStateLabelsWithDefaultStates(s,tm)];

```

Listing C.8: The Rascal implementation of translating the relational definitions of a REBEL2 specification to ALLEALLE.

The next part is responsible for translating all the events, type restrictions and transition function to ALLEALLE. Listing C.9 contain these translations.

```

1  str translateConstraints(rebel::checker::ConfigTranslator::Config cfg, set[Spec] spcs, TModel tm,
2  RelMapping rm)
3  = "<configurationConstraints(cfg.finiteTrace)>
4  '<genericTypeConstraints(cfg.finiteTrace)>
5  '<machineFieldTypeConstraints(spcs, tm)>
6  '<eventParamTypeAndMultiplicityConstraints(spcs, tm)>
7  '<allConfigsAreReachable()>
8  '<onlyOneTriggeringEvent(cfg.finiteTrace)>
9  '<noMachineWithoutState(spcs)>
10 '<machineOnlyHasValuesWhenInitialized(spcs, tm)>
11 '<noTransitionsBetweenUnrelatedStates(cfg.finiteTrace)>
12 '<changeSetPredicates(spcs)>
13 '<helperPredicates()>
14 '<translateEventsToPreds(spcs, rm, tm)>
15 '<constructTransitionFunctions(spcs, rm, tm)>
16 '<translateCompleteTransitionFunction(spcs, cfg)>";
17
18 private str configurationConstraints(bool finiteTrace)
19 = "
20 // Constraints for the configuration and ordering relations
21 order in Config[config as cur] x Config[config as nxt]
22 'last = Config \ order[cur->config] // There is only one last configuration
23 '<if (!finiteTrace) {>back in Config
24 'lone back
25 'some loop
26 'loop in last[config as cur] x back[config as nxt]
27 // Loop contains at most one tuple going back from the last configuration to the<>
28 ";
29
30 private str genericTypeConstraints(bool finiteTrace)
31 = "// Generic 'Type' constraints
32 'raisedEvent in (order<if (!finiteTrace) {> + loop<>)> x
33 'allowedTransitions[event] x Instance[instance]
34 'instanceInState in Instance[instance] x Config x State
35 'changedInstance in (order<if (!finiteTrace) {> + loop<>)> x Instance[instance]
36 ";
37
38 private str machineFieldTypeConstraints(set[Spec] specs, TModel tm) {
39   rel[Spec, str] cons = {};
40
41   for (Spec spc <- specs, /Field f <- spc.fields) {
42     if (isPrim(f.tipe, tm)) {
43       cons += <spc, "getCapitalizedSpecName(spc)><getCapitalizedFieldName(f)>[config,instance]
44         'in Config x (Instance |x| <getCapitalizedSpecName(spc)>)[instance]>";
45     } else {
46       cons += <spc, "getCapitalizedSpecName(spc)><getCapitalizedFieldName(f)>
47         'in Config x (Instance |x| <getCapitalizedSpecName(spc)>)[instance] x
48         '(Instance |x| <getSpecOfType(f.tipe, tm)>)[instance-><f.name>]>";
49     }
50   }
51
52   return "// Machine specific 'type' constraints
53     '<for (Spec spc <- cons<0>) {>// For '<spc.name>'
54     '<for (str fc <- cons[spc]) {><fc>
55     '<><>>";
56 }
57
58 private str machineOnlyHasValuesWhenInitialized(set[Spec] spcs, TModel tm) {
59   str cons = "// Specific per machine: In every configuration iff a machine is
60     '// in an initialized state then it must have values\n";
61
62   for (Spec s <- spcs) {
63     cons += "// for <s.name>\n";
64     bool isEmpty = isEmptySpec(s);

```

```

64
65     for (//Field f <- s.fields) {
66         str relName = "<getCapitalizedSpecName(s)><getCapitalizedFieldName(f)>";
67         list[str] fldCons = [];
68
69         switch (<isPrim(f.tipe, tm), isEmpty) {
70             case <_,true>: {
71                 switch(getType(f, tm)) {
72                     case optionalType(_): fldCons += "lone <relName> |x| c |x| inst";
73                     case setType(_): ;
74                     default: fldCons += "one <relName> |x| c |x| inst";
75                 }
76             }
77             case <true,false>: fldCons += "(((c x inst) |x| instanceInState)[state] in initialized
78                 '<=> one <relName> |x| c |x| inst)";
79             case <false,false>: {
80                 fldCons += "(no (((c x inst) |x| instanceInState)[state] & initialized) =>
81                     'no <relName> |x| c |x| inst)";
82
83                 switch(getType(f, tm)) {
84                     case optionalType(_): fldCons += "(((c x inst) |x| instanceInState)[state] in
85                         'initialized => lone <relName> |x| c |x| inst)";
86                     case setType(_): ;
87                     default: fldCons += "(((c x inst) |x| instanceInState)[state] in initialized =>
88                         'one <relName> |x| c |x| inst)";
89                 }
90             }
91         }
92
93         cons += "<intercalate("\n", ["forall c: Config,
94             'inst: (Instance |x| <getCapitalizedSpecName(s)>)[instance] | <fc> | fc <- fldCons])>\n";
95     }
96 }
97
98 return cons;
99 }
100
101 private str eventParamTypeAndMultiplicityConstraints(set[Spec] spcs, TModel tm) {
102     rel[Spec,str] typeCons = {};
103     rel[Spec,str] multCons = {};
104
105     for (Spec spc <- spcs, Event ev <- spc.events, /FormalParam p <- ev.params) {
106         str relName =
107             "ParamEvent<getCapitalizedSpecName(spc)><getCapitalizedEventName(ev)>
108             '<getCapitalizedParamName(p)>";
109
110         if (isPrim(p.tipe, tm)) {
111             typeCons[spc] = "<relName>[cur,nxt] in order + loop";
112             multCons[spc] = "(some (step |x|
113                 'Event<getCapitalizedSpecName(spc)><getCapitalizedEventName(ev)>
114                 '<=> one (step |x| <relName>))";
115         } else {
116             typeCons[spc] = "<relName> in (order + loop) x
117                 '(Instance |x| <p.tipe.tp>)[instance-><p.name>";
118
119             switch(getType(p,tm)) {
120                 case setType(_): mult = multCons[spc] = "(no (step |x|
121                     'Event<getCapitalizedSpecName(spc)><getCapitalizedEventName(ev)>) =>
122                     'no (step |x| <relName>))";
123
124                 case optionalType(_): multCons[spc] = "((some (step |x|
125                     Event<getCapitalizedSpecName(spc)><getCapitalizedEventName(ev)>) =>
126                     'lone (step |x| <relName>)) &&
127                     '(no (step |x| Event<getCapitalizedSpecName(spc)><getCapitalizedEventName(ev)>) =>

```

```

128         'no (step |x| <relName>));";
129
130     default: multCons[spc] = "(some (step |x|
131     'Event<getCapitalizedSpecName(spc)><getCapitalizedEventName(ev)> <=>
132     'one (step |x| <relName>))";
133     }
134   }
135 }
136
137 return "// Specific per event: parameter type and multiplicity constraints
138 '<for (Spec spc <- typeCons<0>) {>// Type constraints for events of <spc.name>
139 '<for (str tc <- typeCons[spc]) {><tc>
140 '&<><>>
141 '<if (multCons != {}) {>// Multiplicity constraints for event parameters
142 '∀ step: (order + loop) |x| raisedEvent | (
143 ' <intercalate(" &&\n", toList(multCons<1>))>
144 ')<>>";
145 }
146
147 private str allConfigsAreReachable()
148 = "// Generic: All configurations are reachable
149 '∀ c: Config \ first | c in (first[config as cur] |x| ^order)[nxt -\> config]
150 '";
151
152 private str onlyOneTriggeringEvent(bool finiteTrace)
153 = "// Generic: Every transition can only happen by exactly one event
154 '∀ o: order<if (!finiteTrace) {> + loop<>> | one o |x| raisedEvent
155 '";
156
157 private str noMachineWithoutState(set[Spec] spcs)
158 = "// Specific: In every configuration all machines have a state IFF
159 'its a machine which is not empty
160 '∀ c: Config, inst: <nonEmptyMachineInstances(spcs)> | one instanceInState |x| c |x| inst
161 '";
162
163 private str nonEmptyMachineInstances(set[Spec] spcs) {
164   list[str] emptyMachines = [];
165   for (Spec s <- spcs, isEmptySpec(s)) {
166     emptyMachines += "<s.name>";
167   }
168
169   if (emptyMachines == []) {
170     return "Instance";
171   } else {
172     return "(Instance \ (<intercalate(" + ", emptyMachines)>) |x| Instance)";
173   }
174 }
175
176 private str noTransitionsBetweenUnrelatedStates(bool finiteTrace)
177 = "// Generic: Transitions are only allowed between if an event is specified between two states
178 '∀ o: (order<if (!finiteTrace) {> + loop<>>) |x| raisedEvent |
179 ' (o[cur as config] |x| instanceInState)[state-\>from] x (o[nxt as config] |x| instanceInState)
180 ' [state-\>to] x o[event] in allowedTransitions
181 '";
182
183 private str changeSetPredicates(set[Spec] spcs)
184 = "// Change set predicates
185 'pred inChangeSet[step: (cur:id, nxt:id), instances: (instance:id)]
186 ' = instances in (changedInstance |x| step)[instance]
187 '
188 'pred notInChangeSet[step: (cur:id, nxt:id), instances: (instance:id)]
189 ' = no instances & (changedInstance |x| step)[instance]
190 '
191 'pred changeSetCanContain[step: (cur:id, nxt:id), instances: (instance:id)]

```



```

192     ' = (changedInstance |x| step)[instance] in instances
193     ' <if (freeInstances != []) {>+ <intercalate(" + ", freeInstances)><>
194     ""
195     when list[str] freeInstances := ["<s.name>_<inst>" |
196         Spec s <- spcs, /Instances instances <- s.instances,
197         /(Instance)'<Id inst>*' <- instances.instances];
198
199
200 private str helperPredicates()
201 = "// Generic predicates
202 'pred forceState[curState: (state:id), nextState: (state:id), raisedEvent: (event:id)]
203 ' = nextState = (curState[state as from] |x| (allowedTransitions |x| raisedEvent))[to->state]
204 '
205 'pred inState[config: (config:id), instance: (instance:id), state: (state:id)]
206 ' = ((instance x config) |x| instanceInState)[state] in state
207 "";
208
209 private bool hasTransitions(Spec s) = /Transition _ := s.states;
210
211 private str translateEventPredicates(AlleAlleSnippet snippets)
212 = "<for (str spc <- snippets.eventPred<0>) {>// Event predicates for '<spc>'
213     '<for (str ep <- snippets.eventPred[spc]) {><ep>
214     '<><>>";
215
216 private str translatePartialTransitionFunctions(AlleAlleSnippet snippets)
217 = "<for (str spc <- snippets.transPred<0>) {>// Transition function for '<spc>'
218     '<snippets.transPred[spc]>
219     '<>>";
220
221 private str translateCompleteTransitionFunction(set[Spec] spcs, Config cfg)
222 = "// Transition function
223     'forall step: order<if (!cfg.finiteTrace) {> + loop<>>| <intercalate(" && ", posTrans)>
224     ""
225     when
226         posTrans := ["possibleTransitions<getCapitalizedSpecName(s)>[step]" |
227             s <- spcs, hasTransitions(s)],
228         posTrans != [];
229
230 private default str translateCompleteTransitionFunction(set[Spec] spcs, Config cfg) = "";
231
232 private bool isFrameEvent(Event e) = "<e.name>" == "__frame";

```

Listing C.9: The algorithm that translates all REBEL2 constraints to ALLEALLE

During the translation of the events the formula's and expressions are translated. Listing C.10 shows these translation rules.

```

1  data Context = ctx(RelMapping rm, TModel tm, set[Spec] allSpecs, bool topLevelLtl,
2     str curRel, str stepRel, int () nxtUniquePrefix);
3
4  Context nextCurRel(Context old) = ctx(old.rm, old.tm, old.allSpecs, old.topLevelLtl,
5     getNextCurRel(old.curRel), old.stepRel, old.nxtUniquePrefix);
6  Context nextStepRel(Context old) = ctx(old.rm, old.tm, old.allSpecs, old.topLevelLtl,
7     old.curRel, getNextStepRel(old.stepRel), old.nxtUniquePrefix);
8  Context nextCurAndStepRel(Context old) = ctx(old.rm, old.tm, old.allSpecs, old.topLevelLtl,
9     getNextCurRel(old.curRel), getNextStepRel(old.stepRel), old.nxtUniquePrefix);
10
11 Context flipTopLevelLtl(Context old) = ctx(old.rm, old.tm, old.allSpecs, false, old.curRel,
12     old.stepRel, old.nxtUniquePrefix);
13
14 Context replaceCurRel(Context old, str newCurRel) = ctx(old.rm, old.tm, old.allSpecs,
15     old.topLevelLtl, "nxt", old.stepRel, old.nxtUniquePrefix);

```

```

16
17 str translate((Formula)'(<Formula f>)', Context ctx) = "(<translate(f,ctx)>");
18 str translate((Formula)'!(<Formula f>)', Context ctx) = "not (<translate(f,ctx)>)";
19
20 str translate((Formula)'<Expr spc>.<Id event><(<Expr ",,")* params>)', Context ctx) {
21   str relOrSync = translateRelExpr(spc, ctx);
22
23   Spec syncedSpec = getSpecByType(spc, ctx.allSpecs, ctx.tm);
24   Event syncedEvent = lookupEventByName("<event>", syncedSpec);
25
26   // Fix synced event param values
27   list[str] actuals = [ctx.stepRel, "<relOrSync><maybeRename(getFieldName(spc,ctx), "instance">)"];
28
29   list[FormalParam] formals = [p | FormalParam p <- syncedEvent.params];
30   list[Expr] args = [a | Expr a <- params];
31
32   for (int i <- [0..size(formals)]) {
33     switch (args[i]) {
34       case (Expr)'<Int ii>': actuals += "__IntConst_<ii>[const_<ii>-><formals[i].name>]";
35       case (Expr)'<StringConstant s>': actuals += "__StrConst_<unquote(s)>[const_<unquote(s)>->
36         '<formals[i].name>]";
37       default: actuals += "<translateRelExpr(args[i], ctx)>
38         '<maybeRename(getFieldName(args[i], ctx), isPrim(formals[i].tipe,ctx.tm) ?
39           "<formals[i].name>" : "instance">";
40     }
41   }
42
43   return "event<getCapitalizedSpecName(syncedSpec)><getCapitalizedEventName(syncedEvent)>
44     '['<intercalate(", ", actuals)>]";
45 }
46
47 str translate((Formula)'<Expr lhs> is <QualifiedName state>', Context ctx) {
48   str specOfLhs = getSpecTypeName(lhs, ctx.tm);
49   str specRel = ctx.rm[lhs@loc].relExpr;
50
51   str stateRel = "";
52   switch ("<state>") {
53     case "initialized" : stateRel = "initialized";
54     case "finalized" : stateRel = "finalized";
55     case "uninitialized" : stateRel = "uninitialized";
56     default: stateRel = "State<capitalize(specOfLhs)><capitalize(replaceAll("<state>", ":", "_"))>";
57   };
58
59   str stepRel = (Expr)'<Expr _>' := lhs ? "nxt" : "cur";
60   return "inState<stepRel>, <specRel><maybeRename(getFieldName(lhs,ctx), "instance">), <stateRel>]";
61 }
62
63
64 str translate((Formula)'eventually <Formula f>', Context ctx) {
65   str s = ctx.topLevelLtl ? "let cur = first | " : "";
66   ctx = flipTopLevelLtl(ctx);
67
68   return "<s>(exists cur: (cur[config as cur] |x| *(order + loop))[nxt->config] |
69     'let step = cur[config as cur] |x| (order + loop), nxt = step[nxt->config] |
70     '<translate(f,ctx)>";
71 }
72
73 str translate((Formula)'always <Formula f>', Context ctx) {
74   str s = ctx.topLevelLtl ? "let cur = first | " : "";
75   ctx = flipTopLevelLtl(ctx);
76
77   return "<s>(forall cur: (cur[config as cur] |x| *(order + loop))[nxt->config] |
78     'let step = cur[config as cur] |x| (order + loop), nxt = step[nxt->config] |
79     '<translate(f,ctx)>";

```

```

80 }
81
82 str translate((Formula)'always-last <Formula f>', Context ctx) {
83   str s = ctx.topLevelLtl ? "let cur = first | " : "";
84   ctx = flipTopLevelLtl(ctx);
85
86   return "<s>(forall cur: (cur[config as cur] |x| *(order + loop))[nxt->config] - last |
87     'let step = cur[config as cur] |x| (order + loop), nxt = step[nxt->config] |
88     '<translate(f,ctx)>');
89 }
90
91 str translate((Formula)'<Formula u> until <Formula r>', Context ctx) {
92   str s = ctx.topLevelLtl ? "let cur = first | " : "";
93   ctx = flipTopLevelLtl(ctx);
94
95   return "<s>
96     ' ((no loop || (exists goal: (cur[config as cur] |x| *order)[nxt->config] |
97       (let cur = goal, step = cur[config as cur] |x| (order + loop),
98         nxt = step[nxt->config] | <translate(r,ctx)>))) =>
99     (exists goal: (cur[config as cur] |x| *order)[nxt->config] |
100    (let cur = goal, step = cur[config as cur] |x| (order + loop),
101    nxt = step[nxt->config] |
102    (forall cur: ((cur[config->cur] |x| *order)[nxt->config] &
103    (goal[config->nxt] |x| ^order)[cur->config]) |
104    (let step = cur[config as cur] |x| (order + loop), nxt = step[nxt->config] |
105    <translate(u,ctx)>))))))
106    && ((not (no loop || (exists goal: (cur[config as cur] |x| *order)[nxt->config] |
107    (let cur = goal, step = cur[config as cur] |x| (order + loop),
108    nxt = step[nxt->config] | <translate(r,ctx)>)))))) =>
109    (exists goal: (cur[config as cur] |x| *(order+loop))[nxt->config] |
110    (let cur = goal, step = cur[config as cur] |x| (order + loop),
111    nxt = step[nxt->config] |
112    (forall cur: ((cur[config->cur] |x| *order)[nxt->config] +
113    (goal[config->nxt] |x| ^order)[cur->config]) &
114    (last[config->cur] |x| *order)[nxt->config]) |
115    (let step = cur[config as cur] |x| (order + loop), nxt = step[nxt->config] |
116    <translate(u,ctx)>))))));";
117 }
118
119 str translate((Formula)'next <Formula f>', Context ctx) {
120   str s = ctx.topLevelLtl ? "let cur = first | " : "";
121   ctx = flipTopLevelLtl(ctx);
122
123   return "<s>(let cur = (cur[config as cur] |x| (order + loop))[nxt->config],
124     ' step = cur[config as cur] |x| (order + loop), nxt = step[nxt->config] |
125     ' some cur && (<translate(f,ctx)>));";
126 }
127
128 str translate((Formula)'first <Formula f>', Context ctx) {
129   return "let cur = first | <translate(f,ctx)>";
130 }
131
132 str translate((Formula)'<TransEvent event> on <Expr var> <WithAssignments? with>', Context ctx) {
133   str spec = getSpecTypeName(var, ctx.tm);
134   str r = translateRelExpr(var, ctx);
135
136   if ((TransEvent)'*' := event) {
137     return "some (raisedEvent |x| <ctx.stepRel> |x| <r>);";
138   }
139
140   set[str] paramConstraints = {};
141   str spc = "";
142   if (specType(str name) := getType(var,ctx.tm)) {
143     spc = name;

```

```

144 } else {
145     throw "Must be of spec type";
146 }
147
148 for ((Assignment)'<Id fld> = <Expr val>' <- \with) {
149     str paramRel = "(ParamEvent<spc><capitalize("&<event">")><capitalize("&<fld">")> |x|
150         '<ctx.stepRel>)[<fld>]";
151
152     if (isPrim(val, ctx.tm)) {
153         AttRes r = translateAttrExpr(val, ctx);
154         paramConstraints += "some ((<paramRel><if (r.rels != {}) {>
155             'x <intercalate(" x ", [&r.rels])><}>) where (<fld> = <r.constraint>))";
156     } else {
157         paramConstraints += "<paramRel> = <translateRelExpr(val,ctx)>";
158     }
159 }
160
161 return "(<for (pc <- paramConstraints) {><pc> && <}>Event<capitalize(spec)><capitalize("&<event">")>
162     'in (raisedEvent |x| <ctx.stepRel> |x| <r>)[event]);";
163 }
164
165 str translate((Formula)'forall <{Decl ", "+ decls> | <Formula form>', Context ctx)
166     = "(forall <intercalate(" ", [&translate(d,ctx) | Decl d <- decls]> | <translate(form,ctx)>);";
167
168 str translate((Formula)'exists <{Decl ", "+ decls> | <Formula form>', Context ctx)
169     = "(exists <intercalate(" ", [&translate(d,ctx) | Decl d <- decls]> | <translate(form,ctx)>);";
170
171 str translate(current:(Decl)'<Id ", "+ ids>: <Expr expr>', Context ctx) {
172     str te = translateRelExpr(expr, ctx);
173     return intercalate(" ", [{"<name>: <te>" | Id name <- ids}];
174 }
175
176 str translate((Formula)'<Expr lhs> in <Expr rhs>', Context ctx) =
177     "(<translateRelExpr(lhs,ctx)>[<getFieldName(lhs,ctx)>-&]>[<getFieldName(rhs,ctx)>] in
178     ' <translateRelExpr(rhs,ctx)>";
179 str translate((Formula)'<Expr lhs> notin <Expr rhs>', Context ctx) =
180     "no (<translateRelExpr(rhs,ctx)> & <translateRelExpr(lhs,ctx)>[<getFieldName(lhs,ctx)>-&]
181     ' <getFieldName(rhs,ctx)>)];";
182
183 str translate((Formula)'<Formula lhs> && <Formula rhs>', Context ctx) =
184     "(<translate(lhs,ctx)> && <translate(rhs,ctx)>";
185 str translate((Formula)'<Formula lhs> || <Formula rhs>', Context ctx) =
186     "(<translate(lhs,ctx)> || <translate(rhs,ctx)>";
187 str translate((Formula)'<Formula lhs> => <Formula rhs>', Context ctx) =
188     "(<translate(lhs,ctx)> => <translate(rhs,ctx)>";
189 str translate((Formula)'<Formula lhs> \<=> <Formula rhs>', Context ctx) =
190     "(<translate(lhs,ctx)> <=> <translate(rhs,ctx)>";
191
192 str translate((Formula)'<Expr expr> = {}', Context ctx) = "no <translateRelExpr(expr, ctx)>";
193 str translate((Formula)'{} = <Expr expr>', Context ctx) = "no <translateRelExpr(expr, ctx)>";
194 str translate((Formula)'<Expr expr> = none', Context ctx) = "no <translateRelExpr(expr, ctx)>";
195 str translate((Formula)'none = <Expr expr>', Context ctx) = "no <translateRelExpr(expr, ctx)>";
196 default str translate((Formula)'<Expr lhs> = <Expr rhs>', Context ctx) =
197     translateEq(lhs, rhs, "=", ctx);
198
199 str translate((Formula)'<Expr expr> != {}', Context ctx) = "some <translateRelExpr(expr, ctx)>";
200 str translate((Formula)'{} != <Expr expr>', Context ctx) = "some <translateRelExpr(expr, ctx)>";
201 str translate((Formula)'<Expr expr> != none', Context ctx) = "some <translateRelExpr(expr, ctx)>";
202 str translate((Formula)'none != <Expr expr>', Context ctx) = "some <translateRelExpr(expr, ctx)>";
203 default str translate((Formula)'<Expr lhs> != <Expr rhs>', Context ctx) =
204     translateEq(lhs, rhs, "!=", ctx);
205
206 str translate((Formula)'<Expr lhs> \< <Expr rhs>', Context ctx) =
207     translateRestrictionEq(lhs, rhs, "\<", ctx);

```

```

208 str translate((Formula)'<Expr lhs> \<= <Expr rhs>', Context ctx) =
209   translateRestrictionEq(lhs, rhs, "\<=", ctx);
210 str translate((Formula)'<Expr lhs> \>= <Expr rhs>', Context ctx) =
211   translateRestrictionEq(lhs, rhs, "\>=", ctx);
212 str translate((Formula)'<Expr lhs> \> <Expr rhs>', Context ctx) =
213   translateRestrictionEq(lhs, rhs, "\>", ctx);
214
215 str translate((Formula)'if <Formula cond> then <Formula then> else <Formula \else>', Context ctx) =
216   translate((Formula)'(<Formula cond> => <Formula then>) &&
217     (!(<Formula cond>) => <Formula \else>)', ctx);
218
219 str translate((Formula)'if <Formula cond> then <Formula then>', Context ctx) =
220   translate((Formula)'(<Formula cond> => <Formula then>)', ctx);
221
222
223 str translate((Formula)'noOp(<Expr expr>)', Context ctx) {
224   return "notInChangeSet[step, <ctx.rm[expr@loc].relExpr>
225     'renameIfNecessary(expr, "instance", ctx)>]";
226 }
227
228 default str translate(Formula f, Context ctx) {
229   throw "No translation function implemented yet for '<f>'";
230 }
231
232 str translateEq(Expr lhs, Expr rhs, str op, Context ctx) {
233   // Is it equality on attributes?
234   if (isPrim(lhs, ctx.tm) && isPrim(rhs, ctx.tm)) {
235     // it is equality on attributes
236     return translateRestrictionEq(lhs, rhs, op, ctx);
237   } else {
238     return translateRelEq(lhs, rhs, op, ctx);
239   }
240 }
241
242 str translateRelEq(Expr lhs, Expr rhs, str op, Context ctx)
243   = "<translateRelExpr(lhs, ctx)> <op> <translateRelExpr(rhs, ctx)>
244     ' <maybeRename(getFieldName(rhs, ctx), getFieldName(lhs, ctx))>";
245
246 str translateRestrictionEq(Expr lhs, Expr rhs, str op, Context ctx) {
247   AttRes l = translateAttrExpr(lhs, ctx);
248   AttRes r = translateAttrExpr(rhs, ctx);
249
250   return "(some (<intercalate(" x ", [*(l.rels + r.rels)])>) where
251     ' (<l.constraint> <op> <r.constraint>))";
252 }
253
254 str translateRelExpr(current:(Expr)'<Expr e>', Context ctx) = "<translateRelExpr(e, ctx)>";
255 str translateRelExpr(current:(Expr)'<Id id>', Context ctx) = ctx.rm[current@loc].relExpr;
256 str translateRelExpr(current:(Expr)'<Expr expr>', Context ctx) = ctx.rm[current@loc].relExpr;
257 str translateRelExpr(current:(Expr)'<Expr expr>.<Id field>', Context ctx) =
258   ctx.rm[current@loc].relExpr;
259 str translateRelExpr(current:(Expr)'<Expr spc>[<Id field>]', Context ctx) =
260   ctx.rm[current@loc].relExpr;
261
262 str translateRelExpr(current:(Expr)'<Expr expr>.<Id field>', Context ctx) =
263   ctx.rm[current@loc].relExpr;
264 str translateRelExpr(current:(Expr)'<Expr expr>.*<Id field>', Context ctx) =
265   ctx.rm[current@loc].relExpr;
266
267 str translateRelExpr(current:(Expr)'<Id var> : <Expr expr> | <Formula f>', Context ctx) {
268   str te = ctx.rm[expr@loc].relExpr;
269   return "{<var> : <te> | <translate(f, ctx)>}";
270 }
271
272 str translateRelExpr(current:(Expr)'<Expr lhs> + <Expr rhs>', Context ctx) =

```

```

272     ctx.rm[current@\loc].relExpr;
273 str translateRelExpr(current:(Expr)'<Expr lhs> - <Expr rhs>', Context ctx) =
274     ctx.rm[current@\loc].relExpr;
275 str translateRelExpr(current:(Expr)'<{Expr " ,"}* elems>' , Context ctx) =
276     ctx.rm[current@\loc].relExpr;
277
278 str translateRelExpr(current:(Expr)'this', Context ctx) = ctx.rm[current@\loc].relExpr;
279
280 default str translateRelExpr(Expr e, Context ctx) {
281     throw "Can not translate expression '<e>' at location <e@\loc>";
282 }
283
284 alias AttRes = tuple[set[str] rels, str constraint];
285
286 AttRes translateAttrExpr((Expr)'<Expr e>', Context ctx) {
287     AttRes r = translateAttrExpr(e, ctx);
288     return <r.rels, "<r.constraint>">;
289 }
290
291 AttRes translateAttrExpr((Expr)'<Expr e>''', Context ctx) {
292     AttRes r = translateAttrExpr(e, replaceCurRel(ctx, "nxt"));
293     return <r.rels, "<r.constraint>">;
294 }
295
296 AttRes translateAttrExpr(current:(Expr)'<Id id>', Context ctx) {
297     str fld = "param_<ctx.nxtUniquePrefix()>_<id>";
298     str r = "<ctx.rm[current@\loc].relExpr><renameIfNecessary(current, fld, ctx)>";
299     return <{r}, fld>;
300 }
301
302 AttRes translateAttrExpr(current:(Expr)'this.<Id id>', Context ctx) {
303     str r = "<ctx.rm[current@\loc].relExpr><renameIfNecessary(current, "<ctx.curRel>_<id>", ctx)>";
304     return <{r}, "<ctx.curRel>_<id>">;
305 }
306
307 AttRes translateAttrExpr(current:(Expr)'<Expr spc>[<Id inst>].<Id fld>', Context ctx) {
308     str fldName = "<inst>_<fld>_<ctx.nxtUniquePrefix()>";
309     str r = "<ctx.rm[current@\loc].relExpr><renameIfNecessary(current, fldName, ctx)>";
310     return <{r}, fldName>;
311 }
312
313 AttRes translateAttrExpr(current:(Expr)'<Expr expr>.<Id fld>', Context ctx) {
314     str r = ctx.rm[current@\loc].relExpr;
315
316     if (getType(expr, ctx.tm) == stringType() && "<fld>" == "length") {
317         str newFld = "<getFieldName(expr,ctx)>_<ctx.nxtUniquePrefix()>";
318         r = "<r><renameIfNecessary(expr, newFld, ctx)>";
319         return <{r}, "length(<newFld>)">;
320     } else {
321         IdRole role = getIdRole(expr,ctx.tm);
322         str newFld = "<fld>";
323         switch (role) {
324             case fieldId(): newFld = "<ctx.curRel>_<ctx.nxtUniquePrefix()>_<fld>";
325             case paramId(): newFld = "param_<ctx.nxtUniquePrefix()>_<fld>";
326             case quantVarId(): newFld = "<expr>_<fld>_<ctx.nxtUniquePrefix()>";
327         }
328
329         r = "<r><renameIfNecessary(current, newFld, ctx)>";
330         return <{r}, newFld>;
331     }
332 }
333
334 AttRes translateAttrExpr(current:(Expr)'<Id func>(<{Expr " ,"}* actuals>)', Context ctx) {
335     str r = ctx.rm[current@\loc].relExpr;

```

```

336 list[Expr] params = [p | p <- actuals];
337
338 switch("<func>") {
339   case "substr": {
340     str newFld = "<getFieldName(params[0],ctx)>.<ctx.nxtUniquePrefix()>";
341     AttRes sub = translateAttrExpr(params[0],ctx);
342     AttRes frm = translateAttrExpr(params[1],ctx);
343     AttRes to = translateAttrExpr(params[2],ctx);
344
345     r = "<r><renameIfNecessary(params[0], newFld, ctx)>";
346     return <{r} + sub.rels + frm.rels + to.rels,
347           "substr(<newFld>,<frm.constraint>,<to.constraint>)">;
348   }
349   case "toInt": {
350     str newFld = "<getFieldName(params[0],ctx)>.<ctx.nxtUniquePrefix()>";
351     r = "<r><renameIfNecessary(params[0], newFld, ctx)>";
352     AttRes p0 = translateAttrExpr(params[0],ctx);
353     return <{r} + p0.rels, "toInt(<p0.constraint>)">;
354   }
355   case "toStr": {
356     str newFld = "<getFieldName(params[0],ctx)>.<ctx.nxtUniquePrefix()>";
357     r = "<r><renameIfNecessary(params[0], newFld, ctx)>";
358     AttRes p0 = translateAttrExpr(params[0],ctx);
359     return <{r} + p0.rels, "toStr(<p0.constraint>)">;
360   }
361   default: throw "Unknown function '<func>'";
362 }
363 }
364
365 AttRes translateAttrExpr((Expr)'<Expr e>', Context ctx) {
366   AttRes r = translateAttrExpr(e,ctx);
367   return <r.rels, "(<- <r.constraint>)">;
368 }
369
370 AttRes translateAttrExpr(cur:(Expr)'<Expr e>|', Context ctx) {
371   AType tipe = getType(e, ctx.tm);
372   AttRes r = translateAttrExpr(e,ctx);
373
374   if (intType() := tipe) {
375     return <r.rels, "(<r.constraint>)">;
376   } else if (setType(_) := tipe || specType(_) := tipe) {
377     str sizeParam = "size_<ctx.nxtUniquePrefix()>";
378     return <{"(<intercalate(" x ", [re | re <- r.rels])>)[count() as <sizeParam>]"}, sizeParam>;
379   }
380
381   throw "Unable to translate '<e>|' of type '<tipe>'";
382 }
383
384 private AttRes translateBinAttrExpr(Expr lhs, Expr rhs, str op, Context ctx) {
385   AttRes l = translateAttrExpr(lhs,ctx);
386   AttRes r = translateAttrExpr(rhs,ctx);
387
388   return <l.rels + r.rels, "<l.constraint> <op> <r.constraint>">;
389 }
390
391 AttRes translateAttrExpr((Expr)'<Expr lhs> * <Expr rhs>', Context ctx) =
392   translateBinAttrExpr(lhs, rhs, "*", ctx);
393 AttRes translateAttrExpr((Expr)'<Expr lhs> / <Expr rhs>', Context ctx) =
394   translateBinAttrExpr(lhs, rhs, "/", ctx);
395 AttRes translateAttrExpr((Expr)'<Expr lhs> + <Expr rhs>', Context ctx) =
396   translateBinAttrExpr(lhs, rhs, "+", ctx);
397 AttRes translateAttrExpr((Expr)'<Expr lhs> - <Expr rhs>', Context ctx) =
398   translateBinAttrExpr(lhs, rhs, "-", ctx);
399 AttRes translateAttrExpr((Expr)'<Expr lhs> % <Expr rhs>', Context ctx) =

```

```

400     translateBinAttrExpr(lhs, rhs, "%", ctx);
401
402 AttRes translateAttrExpr((Expr)'<Expr lhs> ++ <Expr rhs>', Context ctx) {
403     AttRes l = translateAttrExpr(lhs,ctx);
404     AttRes r = translateAttrExpr(rhs,ctx);
405
406     return <l.rels + r.rels, "<l.constraint> ++ <r.constraint>";
407 }
408
409 AttRes translateAttrExpr(current:(Expr)'<Id var> : <Expr expr> | <Formula f>', Context ctx) {
410     str te = ctx.rm[expr@\loc].relExpr;
411     return <{"<var> : <te> | <translate(f,ctx)>"}", ">";
412 }
413
414 AttRes translateAttrExpr((Expr)'<Lit l>', Context ctx) = <{}, translateLit(l)>;
415
416 default AttRes translateAttrExpr(Expr e, Context ctx) {
417     throw "Can not translate expression '<e>' at location <e@\loc>";
418 }
419
420 str translateLit((Lit)'<Int i>') = "<i>";
421 str translateLit((Lit)'<StringConstant s>') = "<s>";
422
423 str prefix(RelExpr r, str prefix) {
424     if (size(r.heading) > 1) {
425         throw "Can only prefix an unary relation";
426     }
427
428     str fld = getOneFrom(r.heading);
429     return "<r.relExpr><maybeRename(fld, "<prefix>_<fld>")>";
430 }
431
432 str renameIfNecessary(Expr expr, str renamed, Context ctx) {
433     str origName = getFieldName(expr,ctx);
434     if (origName != renamed) {
435         return "[<origName> as <renamed>]";
436     } else {
437         return "";
438     }
439 }
440
441 str getFieldName(Expr expr, Context ctx) {
442     Heading header = ctx.rm[expr@\loc].heading;
443     if (size(header) > 1) {
444         throw "More than 1 attribute in the relation, unable to determine field name";
445     }
446
447     return getOneFrom(header);
448 }
449
450 str getNextCurRel(str oldCurRel) {
451     if (oldCurRel == defaultCurRel()) {
452         return "<defaultCurRel()>_1";
453     }
454
455     if (/cur_<n:[0-9]+>/ := oldCurRel) {
456         return "<defaultCurRel()>_<toInt(n)+1>";
457     }
458 }
459
460 str getNextStepRel(str oldStepRel) {
461     if (oldStepRel == defaultStepRel()) {
462         return "<defaultStepRel()>_1";
463     }

```



```

464
465     if (/step-<n:[0-9]+>/ := oldStepRel) {
466         return "<defaultStepRel(>_<toInt(n)+1>";
467     }
468 }
469
470 str defaultCurRel() = "cur";
471 str defaultStepRel() = "step";
472
473 private str maybeRename(str orig, str renameAs) = "[<orig> as <renameAs>]" when orig != renameAs;
474 private default str maybeRename(str orig, str renameAs) = "";
475
476 str getSpecTypeName(Expr expr, TModel tm) = name when specType(str name) := getType(expr, tm);
477 str getSpecTypeName(Expr expr, TModel tm) =
478     name when optionalType(specType(str name)) := getType(expr, tm);
479
480 default str getSpecTypeName(Expr expr, TModel tm) {
481     throw "Expression '<expr>' is not a Spec Type";
482 }

```

Listing C.10: The algorithm that translates Formula's and Expressions's to ALLEALLE.

Lastly, Listing C.11 contains the translation helper functions that are commonly used in the translation of REBEL2 to ALLEALLE.

```

1  data State
2    = uninitialized()
3    | finalized()
4    | state(str name)
5    | anyState()
6    | noState();
7
8  alias AlleAlleSnippet = tuple[rel[str,str] typeCons, rel[str,str] fieldMultiplicityCons,
9  rel[str,str] paramMultiplicityCons, rel[str,str] eventPred, map[str,str] transPred,
10 rel[str,str] facts, map[str,str] asserts];
11
12 str getLowerCaseSpecName(Spec spc) = toLowerCase("<spc.name>");
13 str getCapitalizedSpecName(Spec spc) = capitalize("<spc.name>");
14 str getCapitalizedEventName(Event e) = capitalize("<e.name>");
15 str getCapitalizedParamName(Param p) = capitalize("<p.name>");
16 str getCapitalizedFieldName(Field f) = capitalize("<f.name>");
17
18 list[str] getInstancesOfType(Type tipe, rel[Spec spc, str instance] instances, TModel tm)
19 = ["<i.instance>" | i <- instances, "<i.spc.name>" == getSpecOfType(tipe, tm)];
20
21 str getSpecOfType(Type tipe, TModel tm) {
22     if (setType(specType(str spc)) := getType(tipe, tm)) {
23         return spc;
24     } else if (specType(str spc) := getType(tipe, tm)) {
25         return spc;
26     } else if (optionalType(specType(str spc)) := getType(tipe, tm)) {
27         return spc;
28     } else {
29         throw "<tipe> is not a (set) spec type";
30     }
31 }
32
33 str type2Str(intType()) = "int";
34 str type2Str(stringType()) = "str";
35 default str type2Str(AType t) = "id";
36
37 str convertType((Type)'Integer') = "int";
38 str convertType((Type)'String') = "str";

```

```

39 default str convertType(Type t) = "id";
40
41 AType getType(Field f, TModel tm) = tm.facts[f.name@\loc] when f.name@\loc in tm.facts;
42 default AType getType(Field f, TModel tm) { throw "No type info available for '<f>'"; }
43
44 AType getType(Expr expr, TModel tm) = tm.facts[expr@\loc] when expr@\loc in tm.facts;
45 default AType getType(Expr expr, TModel tm) {
46     throw "No type info available for '<expr>' at '<expr@\loc>'";
47 }
48
49 AType getIdType(Id id, TModel tm) = tm.facts[id@\loc] when id@\loc in tm.facts;
50 default AType getIdType(Id id, TModel tm) {
51     throw "No type info available for '<id>' at '<id@\loc>'";
52 }
53
54 AType getType(FormalParam p, TModel tm) = tm.facts[p.name@\loc] when p.name@\loc in tm.facts;
55 default AType getType(FormalParam p, TModel tm) {
56     throw "No type info available for '<p>'";
57 }
58
59 AType getType(Type t, TModel tm) = tm.facts[t@\loc] when t@\loc in tm.facts;
60 default AType getType(Type t, TModel tm) {
61     throw "No type info available for '<t>'";
62 }
63
64 IdRole getIdRole(Expr expr, TModel tm) {
65     switch (expr) {
66         case c:(Expr)'this': return getIdRole(c@\loc, tm);
67         case (Expr)'<Id id>': return getIdRole(id@\loc, tm);
68         case (Expr)'<Expr expr>.<Id id>': return getIdRole(id@\loc, tm);
69     }
70
71     throw "No fetch of Id role defined for '<expr>'";
72 }
73
74 IdRole getIdRole(loc expr, TModel tm) = tm.definitions[def].idRole
75 when {loc def} := tm.useDef[expr];
76
77 default IdRole getIdRole(loc expr, TModel tm) {
78     throw "Role can not be found for expression at '<expr>'";
79 }
80
81 bool isParam(Expr expr, TModel tm) = getIdRole(expr,tm) == paramId();
82 default bool isParam(Expr _, TModel _) = false;
83
84 Spec getSpecByType(Expr expr, set[Spec] specs, TModel tm) {
85     AType tipe = getType(expr, tm);
86
87     if (specType(str specName) := tipe || optionalType(specType(str specName)) := tipe) {
88         return lookupSpecByName(specName, specs);
89     }
90
91     throw "Expression '<expr>' is not of spec type";
92 }
93
94 set[Spec] lookupSpecs(rel[Spec spc, str instance, State initialState] instances) =
95     {i.spc | i <- instances};
96
97 private Spec lookupSpecByName(str specName, set[Spec] specs) {
98     for (s <- specs, "<s.name>" == specName) {
99         return s;
100     }
101
102     throw "Spec '<specName>' could not be found";

```

```

103 }
104
105 set[str] lookupStates(Spec spc, TModel tm) {
106   set[str] states = {};
107   for (Define d <- tm.defines, d.idRole == stateId(), d.scope == spc@loc,
108     d.id notin {"initialized","finalized","uninitialized"}) {
109     states += d.id;
110   }
111
112   return states;
113 }
114
115 set[str] lookupStateLabels(Spec spc, TModel tm) =
116   {getStateLabel(spc, st) | str st <- lookupStates(spc,tm)};
117
118 set[str] lookupStateLabelsWithDefaultStates(Spec spc, TModel tm) {
119   set[str] states = lookupStateLabels(spc,tm);
120
121   if /(Transition)'(*) -\> <State _> : <{TransEvent "+ "+ _>;' := spc.states) {
122     states += "state_uninitialized";
123   }
124
125   if /(Transition)'<State _> -\> (*) : <{TransEvent "+ "+ _>;' := spc.states) {
126     states += "state_finalized";
127   }
128
129   return states;
130 }
131
132 str getStateLabel(Spec spc, str state) = "state_<getLowerCaseSpecName(spc)>.<toLowerCase(state)>";
133
134 bool isEmptySpec(Spec spc) = /Transition _ !:= spc.states;
135
136 set[str] lookupInstances(Spec spc, rel[Spec spc, str instance] instances) = instances[spc];
137
138 set[str] lookupEventNames(Spec spc)
139   = {"event_<specName>.<ev>" | Event event <- lookupEvents(spc),
140     str ev := toLowerCase(replaceAll("<event.name>", ":", "_"))}
141   when str specName := toLowerCase("<spec.name>");
142
143 set[str] lookupRaisableEventName(Spec spc)
144   = {"event_<specName>.<ev>" | Event event <- lookupEvents(spc),
145     !isInternalEvent(event), str ev := toLowerCase(replaceAll("<event.name>", ":", "_"))}
146   when str specName := toLowerCase("<spec.name>");
147
148 set[Event] lookupEvents(Spec spc) = {e | /Event e := spc.events};
149
150 Event lookupEventByName(str eventName, Spec spc) {
151   for (Event e <- lookupEvents(spc), "<e.name>" == eventName) {
152     return e;
153   }
154
155   throw "Event with name '<eventName>' could not be found";
156 }
157
158 bool isNonOptionalScalar(Type tipe, TModel tm) = isNonOptionalScalar(t)
159   when tipe@loc in tm.facts, AType t := tm.facts[tipe@loc];
160 default bool isNonOptionalScalar(Type tipe, TModel tm) {
161   throw "No type information found for '<tipe>'";
162 }
163
164 bool isNonOptionalScalar(setType(_)) = false;
165 bool isNonOptionalScalar(optionalType(_)) = false;
166 default bool isNonOptionalScalar(AType _) = true;

```

```

167
168 bool isSetOfInt(Type tipe, TModel tm) = isSetOfType(tipe, intType(), tm);
169 bool isSetOfString(Type tipe, TModel tm) = isSetOfType(tipe, stringType(), tm);
170
171 bool isSetOfPrim(Type tipe, TModel tm) = isSetOfInt(tipe, tm) || isSetOfString(tipe, tm);
172
173 private bool isSetOfType(Type tipe, AType elemType, TModel tm) {
174     if (tipe@\loc notin tm.facts) {
175         throw "No type information found for '<tipe>'";
176     }
177
178     return setType(elemType) := tm.facts[tipe@\loc];
179 }
180
181 bool isPrim(Type tipe, TModel tm) = isPrim(t)
182     when tipe@\loc in tm.facts, AType t := tm.facts[tipe@\loc];
183 default bool isPrim(Type tipe, TModel tm) {
184     throw "No type information found for '<tipe>'";
185 }
186
187 bool isPrim(Expr expr, TModel tm) = isPrim(t)
188     when expr@\loc in tm.facts, AType t := tm.facts[expr@\loc];
189 default bool isPrim(Expr expr, TModel tm) {
190     throw "No type information found for '<expr>' at <expr@\loc>";
191 }
192
193 bool isPrim(intType()) = true;
194 bool isPrim(stringType()) = true;
195 default bool isPrim(AType _) = false;
196
197 bool isInternalEvent(TransEvent te, Spec s) = isInternalEvent(lookupEventByName("<te>", s), s);
198 default bool isInternalEvent(TransEvent te, Spec s) {
199     throw "Unable to find event with name '<te>' in '<s.name>'";
200 }
201
202 bool isInternalEvent(Event e) = /(Modifier)'internal' := e.modifiers;

```

Listing C.11: Translation functions that are commonly used during the translation of REBEL2 to ALLEALLE.

EXAMPLES

d.1 EXAMPLE TRANSLATION FROM REBEL2 TO ALLEALLE

This appendix shows the different intermediate steps when converting the Account running example of Chapter 5. These steps are automatic and generate either a new REBEL 2 specification or an ALLEALLE specification.

Listing D.1 shows the Account specification as originally written by the user. Listing D.2 show the Account specification after applying the first step of the transformation pipeline, the outcome after applying the ‘forget’ and ‘mock’ algorithms (see Listing C.1 for an overview of the ‘forget’ algorithm). Listing D.3 show the result after applying normalization.

After normalization the specification is translated to ALLEALLE. Listing D.4 and Listing D.5 show the first part of the ALLEALLE specification, the definitions of the relations. Listing D.6 and D.7 contain the definition of the type and multiplicity constraints as well as generic helper predicates. Listing D.8, D.9 and D.10 show the encoding of the events as ALLEALLE predicates. Finally, Listing D.11 contains the generated transition function, assertion and optimization criteria.

```

1  module paper::example::Account
2  import paper::example::AccountNumber
3  import paper::example::Date
4
5  spec Account
6      nr: AccountNumber,
7      balance: Integer,
8      openedOn: Date;
9
10     init event open(nr: AccountNumber, openedOn: Date)
11         post: this.nr' = nr, this.balance' = 0, this.openedOn' = openedOn;
12
13     event deposit(amount: Integer)
14         pre: amount > 0;
15         post: this.balance' = this.balance + amount;
16
17     event withdraw(amount: Integer)
18         pre: amount > 0, this.balance >= amount;
19         post: this.balance' = this.balance - amount;
20
21     event payInterest(rate: Integer)
22         post: this.balance' = this.balance + ((this.balance * rate) / 100);
23
24     event block()
25     event unblock()
26     final event forceClose()
27     final event close()
28         pre: this.balance = 0;
29
30     assume AllAccountsHaveUniqueAccountNumbers
31         = always forall ac1, ac2: Account |
32             (ac1 is initialized && ac2 is initialized && ac1.nr = ac2.nr => ac1 = ac2);
33
34     states:
35         (*) -> opened: open;
36         opened -> opened: deposit, withdraw, payInterest;
37         opened -> blocked: block;
38         blocked -> opened: unblock;
39         blocked -> (*): forceClose;
40         opened -> (*): close;
41
42     assert CantOverdrawAccount = always forall a:Account | (a is initialized => a.balance >= 0);
43
44     config Sliced = ac: Account forget nr, openedOn is uninitialized;
45
46     check CantOverdrawAccount from Sliced in max 5 steps;

```

Listing D.1: Original REBEL2 specification of an Account as written by the user.

```

1  module paper::example::Account_CantOverdrawAccount
2
3  spec Account
4      balance: Integer;
5
6      init event open()
7          post: this.balance' = 0;
8
9      event deposit(amount: Integer)
10         pre: amount > 0;
11         post: this.balance' = this.balance + amount;
12
13     event withdraw(amount: Integer)
14         pre: amount > 0, this.balance >= amount;
15         post: this.balance' = this.balance - amount;
16
17     event payInterest(rate: Integer)
18         post: this.balance' = this.balance + ((this.balance * rate) / 100);
19
20     event block()
21     event unblock()
22     final event forceClose()
23
24     final event close()
25         pre: this.balance = 0;
26
27     states:
28         (*) -> opened: open;
29         opened -> opened: deposit, withdraw, payInterest;
30         opened -> blocked: block;
31         blocked -> opened: unblock;
32         blocked -> (*): forceClose;
33         opened -> (*): close;
34
35     assert CantOverdrawAccount = !(always forall a:Account | (a is initialized => a.balance >= 0));
36
37     config Sliced = ac : Account is uninitialized ;
38
39     check CantOverdrawAccount from Sliced in max 5 steps;

```

Listing D.2: Generated REBEL2 specification after applying the ‘forget’ and ‘mock’ algorithms. In this case only ‘forget’ was needed since the configuration used (Sliced, line 44 of Listing D.1) specified that both the nr and openedOn fields are to be forgotten but no specifications were instructed to be mocked. Please also notice that the assertion (CantOverdrawAccount) has been negated. Since a check is performed, we are interested in finding a state where the assertion does not hold.

```

1  module paper::example::Account_CantOverdrawAccount
2
3  spec Account
4      balance: Integer;
5
6      init event open()
7          post: this.balance' = 0;
8
9      event deposit(amount: Integer)
10         pre: amount > 0;
11         post: this.balance' = this.balance + amount;
12
13     event withdraw(amount: Integer)
14         pre: amount > 0, this.balance >= amount;
15         post: this.balance' = this.balance - amount;
16
17     event payInterest(rate: Integer)
18         post: this.balance' = this.balance + ((this.balance * rate) / 100);
19
20     event block()
21         post: this.balance' = this.balance;
22
23     event unblock()
24         post: this.balance' = this.balance;
25
26     final event forceClose()
27
28     final event close()
29         pre: this.balance = 0;
30
31     internal event __frame()
32         post: this.balance' = this.balance;
33
34     states:
35         (*) -> opened : open;
36         opened -> opened : deposit;
37         opened -> opened : withdraw;
38         opened -> opened : payInterest;
39         opened -> blocked : block;
40         blocked -> opened : unblock;
41         blocked -> (*) : forceClose;
42         opened -> (*) : close;
43
44     assert CantOverdrawAccount = !(always forall a:Account | (a is initialized => a.balance >= 0));
45
46     config Sliced = ac : Account is uninitialized;
47
48     check CantOverdrawAccount from Sliced in max 5 steps;

```

Listing D.3: Generated REBEL2 specification after normalization. Frame conditions were added to the event as well as a new `__frame()` event which fixes all values between steps.


```

1 // Define the specs that can take place in the transition system
2 Account (spec:id) = {<account>}
3
4 // Define all possible states for all machines
5 State (state:id) = {<state_account_opened>,<state_account_blocked>,<state_uninitialized>,<state_finalized>}
6
7 initialized (state:id) = {<state_account_opened>,<state_account_blocked>}
8 finalized (state:id) = {<state_finalized>}
9 uninitialized (state:id) = {<state_uninitialized>}
10 StateAccountOpened (state:id) = {<state_account_opened>}
11 StateAccountBlocked (state:id) = {<state_account_blocked>}
12
13 // Define which transitions are allowed in the form of:
14 // 'from a state' -> ' via an event' -> 'to a state'
15 allowedTransitions (from:id, to:id, event:id) = {
16   <state_account_opened,state_account_opened,event_account_deposit>,
17   <state_account_opened,state_account_blocked,event_account_block>,
18   <state_account_opened,state_finalized,event_account_close>,
19   <state_account_opened,state_account_opened,event_account_withdraw>,
20   <state_account_blocked,state_account_opened,event_account_unblock>,
21   <state_account_blocked,state_finalized,event_account_forceclose>,
22   <state_uninitialized,state_account_opened,event_account_open>,
23   <state_account_opened,state_account_opened,event_account_payinterest>}
24
25 // Define each event as single relation so that the events can be used as variables in the constraints
26 EventAccountForceClose (event:id) = {<event_account_forceclose>}
27 EventAccountWithdraw (event:id) = {<event_account_withdraw>}
28 EventAccountOpen (event:id) = {<event_account_open>}
29 EventAccountDeposit (event:id) = {<event_account_deposit>}
30 EventAccountPayInterest (event:id) = {<event_account_payinterest>}
31 EventAccount__frame (event:id) = {<event_account___frame>}
32 EventAccountUnblock (event:id) = {<event_account_unblock>}
33 EventAccountBlock (event:id) = {<event_account_block>}
34 EventAccountClose (event:id) = {<event_account_close>}

```

Listing D.4: Generated ALLEALLE relation definition for the 'static' part of the Account specification

```

1 Config (config: id)      >= {<c1>} <= {<c1>, <c2>, <c3>, <c4>, <c5>, <c6>}
2
3 order (cur: id, nxt: id) <= {<c1, c2>, <c2, c3>, <c3, c4>, <c4, c5>, <c5, c6>}
4 first (config: id)      = {<c1>}
5 last (config: id)       <= {<c1>, <c2>, <c3>, <c4>, <c5>, <c6>}
6 back (config: id)      = {}
7 loop (cur: id, nxt: id) = {}
8
9 Instance (spec: id, instance: id) = {<account, ac>}
10
11 instanceInState (config: id, instance: id, state: id) >= {<c1, ac, state_uninitialized>} <= {
12   <c1, ac, state_account_opened>, <c1, ac, state_account_blocked>, <c1, ac, state_uninitialized>,
13   <c1, ac, state_finalized>, <c2, ac, state_account_opened>, <c2, ac, state_account_blocked>,
14   <c2, ac, state_uninitialized>, <c2, ac, state_finalized>, <c3, ac, state_account_opened>,
15   <c3, ac, state_account_blocked>, <c3, ac, state_uninitialized>, <c3, ac, state_finalized>,
16   <c4, ac, state_account_opened>, <c4, ac, state_account_blocked>, <c4, ac, state_uninitialized>,
17   <c4, ac, state_finalized>, <c5, ac, state_account_opened>, <c5, ac, state_account_blocked>,
18   <c5, ac, state_uninitialized>, <c5, ac, state_finalized>, <c6, ac, state_account_opened>,
19   <c6, ac, state_account_blocked>, <c6, ac, state_uninitialized>, <c6, ac, state_finalized>}
20
21 raisedEvent (cur: id, nxt: id, event: id, instance: id) <= {
22   <c1, c2, event_account_forceclose, ac>, <c1, c2, event_account_deposit, ac>,
23   <c1, c2, event_account_block, ac>, <c1, c2, event_account_payinterest, ac>,
24   <c1, c2, event_account_unblock, ac>, <c1, c2, event_account_close, ac>, <c1, c2, event_account_open, ac>,
25   <c1, c2, event_account_withdraw, ac>, <c2, c3, event_account_forceclose, ac>,
26   <c2, c3, event_account_deposit, ac>, <c2, c3, event_account_block, ac>,
27   <c2, c3, event_account_payinterest, ac>, <c2, c3, event_account_unblock, ac>,
28   <c2, c3, event_account_close, ac>, <c2, c3, event_account_open, ac>, <c2, c3, event_account_withdraw, ac>,
29   <c3, c4, event_account_forceclose, ac>, <c3, c4, event_account_deposit, ac>,
30   <c3, c4, event_account_block, ac>, <c3, c4, event_account_payinterest, ac>,
31   <c3, c4, event_account_unblock, ac>, <c3, c4, event_account_close, ac>, <c3, c4, event_account_open, ac>,
32   <c3, c4, event_account_withdraw, ac>, <c4, c5, event_account_forceclose, ac>,
33   <c4, c5, event_account_deposit, ac>, <c4, c5, event_account_block, ac>,
34   <c4, c5, event_account_payinterest, ac>, <c4, c5, event_account_unblock, ac>,
35   <c4, c5, event_account_close, ac>, <c4, c5, event_account_open, ac>, <c4, c5, event_account_withdraw, ac>,
36   <c5, c6, event_account_forceclose, ac>, <c5, c6, event_account_deposit, ac>,
37   <c5, c6, event_account_block, ac>, <c5, c6, event_account_payinterest, ac>,
38   <c5, c6, event_account_unblock, ac>, <c5, c6, event_account_close, ac>, <c5, c6, event_account_open, ac>,
39   <c5, c6, event_account_withdraw, ac>}
40
41 changedInstance (cur: id, nxt: id, instance: id) <= {
42   <c1, c2, ac>, <c2, c3, ac>, <c3, c4, ac>, <c4, c5, ac>, <c5, c6, ac>}
43
44 AccountBalance (config: id, instance: id, balance: int) <= {
45   <c1, ac, ?>, <c2, ac, ?>, <c3, ac, ?>, <c4, ac, ?>, <c5, ac, ?>, <c6, ac, ?>}
46
47 ParamEventAccountWithdrawAmount (cur: id, nxt: id, amount: int) <= {
48   <c1, c2, ?>, <c2, c3, ?>, <c3, c4, ?>, <c4, c5, ?>, <c5, c6, ?>}
49
50 ParamEventAccountPayInterestRate (cur: id, nxt: id, rate: int) <= {
51   <c1, c2, ?>, <c2, c3, ?>, <c3, c4, ?>, <c4, c5, ?>, <c5, c6, ?>}
52
53 ParamEventAccountDepositAmount (cur: id, nxt: id, amount: int) <= {
54   <c1, c2, ?>, <c2, c3, ?>, <c3, c4, ?>, <c4, c5, ?>, <c5, c6, ?>}

```

Listing D.5: Generated ALLEALLE relation definition for the ‘dynamic’ part of the Account specification

```

1 // Constraints for the configuration and ordering relations
2 order in Config[config as cur] x Config[config as nxt]
3 last = Config \ order[cur->config] // There is only one last configuration
4
5 // Generic 'Type' constraints
6 raisedEvent in (order) x allowedTransitions[event] x Instance[instance]
7 instanceInState in Instance[instance] x Config x State
8 changedInstance in (order) x Instance[instance]
9
10 // Machine specific 'type' constraints
11 // - for 'Account'
12 AccountBalance[config,ininstance] in Config x (Instance |x| Account)[ininstance]
13
14 // Specific per event: parameter type and multiplicity constraints
15 // Type constraints for events of Account
16 ParamEventAccountDepositAmount[cur,nxt] in order + loop
17 ParamEventAccountWithdrawAmount[cur,nxt] in order + loop
18 ParamEventAccountPayInterestRate[cur,nxt] in order + loop
19
20 // Multiplicity constraints for event parameters
21 forall step: (order + loop) |x| raisedEvent | (
22   (some (step |x| EventAccountDeposit) <=> one (step |x| ParamEventAccountDepositAmount)) &&
23   (some (step |x| EventAccountPayInterest) <=> one (step |x| ParamEventAccountPayInterestRate)) &&
24   (some (step |x| EventAccountWithdraw) <=> one (step |x| ParamEventAccountWithdrawAmount))
25 )
26 // Generic: All configurations are reachable
27 forall c: Config \ first | c in (first[config as cur] |x| ^order)[nxt -> config]
28
29 // Generic: Every transition can only happen by exactly one event
30 forall o: order | one o |x| raisedEvent
31
32 // Specific: In every configuration all machines have a state IFF its a machine which is not empty
33 forall c: Config, inst: Instance | one instanceInState |x| c |x| inst
34
35 // Specific per machine: In every configuration iff a machine is in an initialized state
36 // then it must have values
37 // - for Account
38 forall c: Config, inst: (Instance |x| Account)[ininstance] |
39   (((c x inst) |x| instanceInState)[state] in initialized <=> one AccountBalance |x| c |x| inst)
40
41 // Generic: Transitions are only allowed between states if an event is specified for those two states
42 forall o: (order) |x| raisedEvent |
43   (o[cur as config] |x| instanceInState)[state->from] x
44   (o[nxt as config] |x| instanceInState)[state->to] x
45   o[event] in allowedTransitions

```

Listing D.6: Generated ALLEALLE constraints encoding the type and multiplicity constraints of the Account specification

```

1 // Allow for an instance to be part of the changing machines
2 pred inChangeSet[step: (cur:id, nxt:id), instances: (instance:id)]
3   = instances in (changedInstance |x| step)[instance]
4
5 // Disallow an instance to be part of the changing machines
6 pred notInChangeSet[step: (cur:id, nxt:id), instances: (instance:id)]
7   = no instances & (changedInstance |x| step)[instance]
8
9 pred changeSetCanContain[step: (cur:id, nxt:id), instances: (instance:id)]
10  = (changedInstance |x| step)[instance] in instances
11
12 pred forceState[curState: (state:id), nextState: (state:id), raisedEvent: (event:id)]
13  = nextState = (curState[state as from] |x| (allowedTransitions |x| raisedEvent))[to->state]
14
15 pred inState[config: (config:id), instance: (instance:id), state: (state:id)]
16  = ((instance x config) |x| instanceInState)[state] in state
17
18 pred frameAccount[step: (cur:id, nxt:id), account: (instance:id)]
19  = let cur = step[cur->config],
20     nxt = step[nxt->config],
21     curState = (instanceInState |x| cur |x| account)[state],
22     nextState = (instanceInState |x| nxt |x| account)[state] | (
23     nextState = curState && (
24     curState in uninitialized ||
25     // Postconditions
26     some ((account |x| (AccountBalance |x| nxt))[balance][balance as nxt_balance] x
27     (account |x| (AccountBalance |x| cur))[balance][balance as cur_balance])
28     where (nxt_balance = cur_balance))
29  )

```

Listing D.7: Generated ALLEALLE constraints encoding helper predicates and the generated frame event which is enforced when an instance is not allow to change during a state transition.

```

1 // Event predicates for Account
2 pred eventAccountOpen[step:(cur:id, nxt:id), account: (instance:id)]
3   = let cur = step[cur->config],
4     nxt = step[nxt->config],
5     curState = (instanceInState |x| cur |x| account)[state],
6     nxtState = (instanceInState |x| nxt |x| account)[state] | (
7     // Postconditions
8     (some ((account |x| (AccountBalance |x| nxt))[balance][balance as nxt_balance])
9     where (nxt_balance = 0)) &&
10    // Generic event conditions
11    forceState[curState, nxtState, EventAccountOpen] &&
12    // Make sure this instance is in the change set
13    inChangeSet[step, account]
14  )
15
16 pred eventAccountDeposit[step:(cur:id, nxt:id), account: (instance:id), amount: (amount:int)]
17   = let cur = step[cur->config],
18     nxt = step[nxt->config],
19     curState = (instanceInState |x| cur |x| account)[state],
20     nxtState = (instanceInState |x| nxt |x| account)[state] | (
21     // Preconditions
22     (some (amount[amount as param_1.amount] where (param_1.amount > 0)) &&
23     // Postconditions
24     (some ((account |x| (AccountBalance |x| nxt))[balance][balance as nxt_balance] x
25     (account |x| (AccountBalance |x| cur))[balance][balance as cur_balance] x
26     amount[amount as param_2.amount] where (nxt_balance = cur_balance + param_2.amount)) &&
27     // Generic event conditions
28     forceState[curState, nxtState, EventAccountDeposit] &&
29     // Make sure this instance is in the change set
30     inChangeSet[step, account]
31  )
32
33 pred eventAccountWithdraw[step:(cur:id, nxt:id), account: (instance:id), amount: (amount:int)]
34   = let cur = step[cur->config],
35     nxt = step[nxt->config],
36     curState = (instanceInState |x| cur |x| account)[state],
37     nxtState = (instanceInState |x| nxt |x| account)[state] | (
38     // Preconditions
39     (some (amount[amount as param_1.amount] where (param_1.amount > 0)) &&
40     (some ((account |x| (AccountBalance |x| cur))[balance][balance as cur_balance] x
41     amount[amount as param_2.amount] where (cur_balance >= param_2.amount)) &&
42     // Postconditions
43     (some ((account |x| (AccountBalance |x| nxt))[balance][balance as nxt_balance] x
44     (account |x| (AccountBalance |x| cur))[balance][balance as cur_balance] x
45     amount[amount as param_3.amount] where (nxt_balance = cur_balance - param_3.amount)) &&
46     // Generic event conditions
47     forceState[curState, nxtState, EventAccountWithdraw] &&
48     // Make sure this instance is in the change set
49     inChangeSet[step, account]
50  )

```

Listing D.8: Generated ALLEALLE constraints encoding the open, deposit and withdraw events of the Account specification

```

1  pred eventAccountPayInterest[step:(cur:id, nxt:id), account: (instance:id), rate: (rate:int)]
2  = let cur = step[cur->config],
3      nxt = step[nxt->config],
4      curState = (instanceInState |x| cur |x| account)[state],
5      nxtState = (instanceInState |x| nxt |x| account)[state] | (
6      // Postconditions
7      (some ((account |x| (AccountBalance |x| nxt))[balance][balance as nxt_balance] x
8              rate[rate as param_1_rate] x
9              (account |x| (AccountBalance |x| cur))[balance][balance as cur_balance])
10             where (nxt_balance = cur_balance + ((cur_balance * param_1_rate) / 100))) &&
11             // Generic event conditions
12             forceState[curState, nxtState, EventAccountPayInterest] &&
13             // Make sure this instance is in the change set
14             inChangeSet[step, account]
15     )
16
17  pred eventAccountBlock[step:(cur:id, nxt:id), account: (instance:id)]
18  = let cur = step[cur->config],
19      nxt = step[nxt->config],
20      curState = (instanceInState |x| cur |x| account)[state],
21      nxtState = (instanceInState |x| nxt |x| account)[state] | (
22      // Postconditions
23      (some ((account |x| (AccountBalance |x| nxt))[balance][balance as nxt_balance] x
24              (account |x| (AccountBalance |x| cur))[balance][balance as cur_balance])
25              where (nxt_balance = cur_balance)) &&
26              // Generic event conditions
27              forceState[curState, nxtState, EventAccountBlock] &&
28              // Make sure this instance is in the change set
29              inChangeSet[step, account]
30     )
31
32  pred eventAccountUnblock[step:(cur:id, nxt:id), account: (instance:id)]
33  = let cur = step[cur->config],
34      nxt = step[nxt->config],
35      curState = (instanceInState |x| cur |x| account)[state],
36      nxtState = (instanceInState |x| nxt |x| account)[state] | (
37      // Postconditions
38      (some ((account |x| (AccountBalance |x| nxt))[balance][balance as nxt_balance] x
39              (account |x| (AccountBalance |x| cur))[balance][balance as cur_balance])
40              where (nxt_balance = cur_balance)) &&
41              // Generic event conditions
42              forceState[curState, nxtState, EventAccountUnblock] &&
43              // Make sure this instance is in the change set
44              inChangeSet[step, account]
45     ) )

```

Listing D.9: Generated ALLEALLE constraints encoding the payInterest, block and unblock events of the Account specification

```

1  pred eventAccountForceClose[step:(cur:id, nxt:id), account: (instance:id)]
2    = let cur = step[cur->config],
3      nxt = step[nxt->config],
4      curState = (instanceInState |x| cur |x| account)[state],
5      nxtState = (instanceInState |x| nxt |x| account)[state] | (
6        // Generic event conditions
7        forceState[curState, nxtState, EventAccountForceClose] &&
8        // Make sure this instance is in the change set
9        inChangeSet[step, account]
10   )
11
12  pred eventAccountClose[step:(cur:id, nxt:id), account: (instance:id)]
13    = let cur = step[cur->config],
14      nxt = step[nxt->config],
15      curState = (instanceInState |x| cur |x| account)[state],
16      nxtState = (instanceInState |x| nxt |x| account)[state] | (
17        // Preconditions
18        (some ((account |x| (AccountBalance |x| cur))[balance][balance as cur_balance])
19          where (cur_balance = 0)) &&
20        // Generic event conditions
21        forceState[curState, nxtState, EventAccountClose] &&
22        // Make sure this instance is in the change set
23        inChangeSet[step, account]
24   )

```

Listing D.10: Generated ALLEALLE constraints encoding the forceClose and close events of the Account specification

```

1 // Transition function for Account
2 pred possibleTransitionsAccount[step: (cur:id, nxt:id)]
3 = forall inst: (Instance |x| Account)[instance] |
4   (some inst & ((raisedEvent |x| step)[instance]) <=> (
5     (eventAccountOpen[step,inst] &&
6       (step |x| raisedEvent)[event] = EventAccountOpen &&
7       changeSetCanContain[step, inst])
8     ||
9     (eventAccountBlock[step,inst] &&
10      (step |x| raisedEvent)[event] = EventAccountBlock &&
11      changeSetCanContain[step, inst])
12     ||
13     (eventAccountClose[step,inst] &&
14      (step |x| raisedEvent)[event] = EventAccountClose &&
15      changeSetCanContain[step, inst])
16     ||
17     (eventAccountForceClose[step,inst] &&
18      (step |x| raisedEvent)[event] = EventAccountForceClose &&
19      changeSetCanContain[step, inst])
20     ||
21     (eventAccountWithdraw[step,inst,(step |x| ParamEventAccountWithdrawAmount)[amount]] &&
22      (step |x| raisedEvent)[event] = EventAccountWithdraw &&
23      changeSetCanContain[step, inst])
24     ||
25     (eventAccountPayInterest[step,inst,(step |x| ParamEventAccountPayInterestRate)[rate]] &&
26      (step |x| raisedEvent)[event] = EventAccountPayInterest &&
27      changeSetCanContain[step, inst])
28     ||
29     (eventAccountUnblock[step,inst] &&
30      (step |x| raisedEvent)[event] = EventAccountUnblock &&
31      changeSetCanContain[step, inst])
32     ||
33     (eventAccountDeposit[step,inst,(step |x| ParamEventAccountDepositAmount)[amount]] &&
34      (step |x| raisedEvent)[event] = EventAccountDeposit &&
35      changeSetCanContain[step, inst])
36   ))
37   &&
38   (notInChangeSet[step, inst] => frameAccount[step, inst])
39
40 // Transition function
41 forall step: order| possibleTransitionsAccount[step]
42
43 // Assert 'CantOverdrawAccount'
44 not ((let cur = first | (forall cur: (cur[config as cur] |x| *(order + loop))[nxt->config] |
45   let step = cur[config as cur] |x| (order + loop), nxt = step[nxt->config] |
46     (forall a: (Instance |x| Account)[instance] |
47       ((inState[cur, a, initialized] =>
48         (some ((a |x| (AccountBalance |x| cur))[balance][balance as a_balance_1])
49           where (a_balance_1 >= 0)))))))
50
51 // Minimize the number of steps by minimizing the number of Configurations
52 objectives: minimize Config[count()]

```

Listing D.11: Generated ALLEALLE constraints encoding the payInterest, block and unblock events of the Account specification

DATA

e.1 OPTIMAL PACKAGE DEPENDENCY RESOLUTION

Table E.1: “MISC paranoid” ALLEALLE results. Problem names refer to the problems as they were named in the original competition. See <http://www.mancoosi.org/misc-2012/results/paranoid/> for an overview.

Problem name	Request type	# of Packages in CUDF	# dependencies	ALLEALLE translation time (in sec)	Z ₃ solving time (in sec)	Best 2012 competition solving time (in sec)	Correct?	Optimal?
adf7b774-9af8-11df-bc37-00163e46d37a	upgrade	47203	22721	554.93	2.88	1.30	yes	yes
eobd67a6-56d0-11df-b11f-00163e7a6f5e	upgrade	28333	14170	343.96	2.29	0.63	yes	yes
e2f6303a-4fe9-11e0-aa4f-00163e1e087d	install	41913	29726	1077.47	1.72	2.57	yes	yes
29180036-5408-11df-9f57-00163e7a6f5e	upgrade	28753	17490	403.31	1.22	0.99	yes	yes
7bf50d1c-9b1b-11df-8b50-00163e46d37a	upgrade	47210	17934	459.80	3.35	1.32	yes	yes
8boe7c16-bab4-11e0-a883-00163e1e087d	install	59100	57811	2879.15	4.20	2.60	yes	yes
5698a62c-c731-11df-9bb9-00163e3d3b7c	install	54474	62705	3463.34	4.71	2.80	yes	yes
6bod1dao-c730-11df-a7c5-00163e3d3b7c	install	54474	62792	3609.27	5.86	1.62	yes	yes
19890cfe-db9f-11df-9e6c-00163e3d3b7c	upgrade	33615	23979	743.49	3.24	1.15	yes	yes
978532fa-c730-11df-b070-00163e3d3b7c	install	54474	62792	3425.55	4.50	2.80	yes	yes
dd08e73e-d489-11df-b9cf-00163e3d3b7c	install	49561	48746	2230.60	3.20	1.78	yes	yes
d1583bd8-d489-11df-9a24-00163e3d3b7c	install	49561	48746	2327.53	3.68	1.44	yes	yes
dba3a3fe-3477-11e0-9e6c-00163e3d3b7c	upgrade	36310	17891	189.83	1.18	0.79	yes	yes
f4ebf9e0-360e-11e0-9e6c-00163e3d3b7c	upgrade	36213	18015	464.63	1.21	1.80	yes	yes
4ede8d96-c17a-11df-a7c5-00163e3d3b7c	install	51849	40000	1630.78	2.45	1.47	yes	yes
33bb2fbc-9512-11e0-9181-00163e1e087d	install	51134	15370	336.95	0.73	1.51	yes	yes
ab9005be-bacc-11e0-b0f6-00163e1e087d	install	59100	57811	2914.14	4.15	2.47	yes	yes
ff4a1d84-d490-11df-9e6c-00163e3d3b7c	install	53540	53307	2519.30	2.40	2.26	yes	yes
fa3d0fb2-db9e-11df-a0ec-00163e3d3b7c	upgrade	33615	23979	726.63	3.14	0.89	yes	yes
80e3fda2-9501-11e0-8001-00163e1e087d	install	51134	15370	342.71	0.74	1.57	yes	yes
d023d256-3477-11e0-bdb2-00163e3d3b7c	upgrade	36310	17891	445.69	2.57	1.10	yes	yes
103c9978-5408-11df-9bc1-00163e7a6f5e	upgrade	28753	17490	440.41	1.21	0.76	yes	yes
4a69cf16-c731-11df-9182-00163e3d3b7c	install	54474	62705	3761.11	4.74	1.62	yes	yes
26f3d4cc-d470-11df-9e6c-00163e3d3b7c	install	49561	48746	2272.93	3.28	1.65	yes	yes
ec32fc68-7254-11e0-8436-00163e1e087d	upgrade	39025	21410	575.76	1.33	1.50	yes	yes
cff2854-9512-11e0-8001-00163e1e087d	install	51134	15370	337.73	0.67	1.20	yes	yes
8680dd8a-8600-11e0-b285-00163e1e087d	install	53360	51425	2427.60	3.92	1.92	yes	yes
caefdef6-3477-11e0-84ef-00163e3d3b7c	upgrade	36310	17891	429.89	1.25	1.26	yes	yes
e381ba7e-a192-11e0-8647-00163e1e087d	install	51134	15370	349.90	0.79	1.54	yes	yes
deb285a6-db9e-11df-8f4f-00163e3d3b7c	upgrade	33615	23979	696.62	2.81	0.93	yes	yes
ca8ff65c-db9e-11df-b9cf-00163e3d3b7c	upgrade	33615	23979	860.79	2.81	0.82	yes	yes
e599f3fc-360e-11e0-986e-00163e3d3b7c	upgrade	36213	18015	466.59	1.26	1.14	yes	yes

Table E.1: “MISC paranoid” ALLEALLE results. Problem names refer to the problems as they were named in the original competition. See <http://www.mancoosi.org/misc-2012/results/paranoid/> for an overview.

Problem name	Request type	# of Packages in CUDF	# dependencies	ALLEALLE translation time (in sec)	Z3 solving time (in sec)	Best 2012 competition solving time (in sec)	Correct?	Optimal?
d0cc7514-c730-11df-a040-00163e3d3b7c	install	54474	62705	3392.81	4.50	2.90	yes	yes
bccf69ae-db9e-11df-9a24-00163e3d3b7c	upgrade	33615	23979	682.21	2.81	0.96	yes	yes
27000e82-c5c4-11df-a7c5-00163e3d3b7c	install	54485	62817	3393.98	4.14	1.34	yes	yes
d5026b8e-3477-11e0-986e-00163e3d3b7c	upgrade	36310	17891	448.12	1.20	0.94	yes	yes
e69a0e36-9ef1-11df-9d4a-00163e46d37a	install	50726	63860	3582.47	4.58	1.68	yes	yes
56e31304-c17a-11df-b070-00163e3d3b7c	install	51849	40000	1559.44	2.21	1.26	yes	yes
ed1cc19e-51b7-11e0-8436-00163e1e087d	install	42104	29996	948.50	1.32	1.66	yes	yes
a754ac72-95cc-11e0-9181-00163e1e087d	install	51134	15370	322.43	0.64	1.12	yes	yes
56ae4afa-ob33-11df-8a2b-00163e1d94dc	install	66940	4085	47.15	0.23	1.91	yes	yes
4e539b28-d46c-11df-8f4f-00163e3d3b7c	install	49561	48746	2309.19	3.20	1.50	yes	yes
fe523ea6-9b1b-11df-bc37-00163e46d37a	upgrade	47210	22707	573.45	3.40	0.95	yes	yes
b2540c52-51b7-11e0-aa4f-00163e1e087d	install	42104	29996	894.50	1.39	1.72	yes	yes
eeee44ce-5407-11df-b11f-00163e7a6f5e	upgrade	28753	17490	410.22	1.22	0.63	yes	yes
80cf9a6-9b1b-11df-965e-00163e46d37a	upgrade	47210	17934	420.83	1.74	1.32	yes	yes
1aabfc32-d491-11df-9a24-00163e3d3b7c	install	53540	53307	2685.60	2.88	2.14	yes	yes
8222799a-9af8-11df-8b50-00163e46d37a	upgrade	47203	17933	422.87	1.70	1.22	yes	yes
301cbe92-a79c-11e0-9181-00163e1e087d	install	56865	55542	2798.53	0.00	0.00	yes	yes
7c834coe-51b8-11e0-a49e-00163e1e087d	install	42104	29996	951.58	1.46	2.58	yes	yes
3e4f8550-ob33-11df-942d-00163e1d94dc	install	66940	4085	60.48	0.68	1.27	yes	yes
8afd89e-51b8-11e0-acd7-00163e1e087d	install	42104	29996	897.15	1.31	1.66	yes	yes
4f84e9c6-a79c-11e0-9eb7-00163e1e087d	install	56865	55542	2785.50	0.00	0.00	yes	yes
7f80e4fo-4fe9-11e0-acd7-00163e1e087d	install	41913	29720	897.63	1.55	1.79	yes	yes
c2164c84-b015-11df-8b50-00163e46d37a	upgrade	32938	11545	236.60	0.33	1.11	yes	yes
0207e19a-9b1c-11df-af69-00163e46d37a	upgrade	47210	22707	571.12	2.67	1.30	yes	yes

SUMMARY

Large enterprises such as banks face many challenges when it comes to controlling the every growing complexity of their systems. These systems are never created in one go, they are the result of many iterations during many decades of development. Since techniques evolve, so do these software systems. Controlling this complexity is a wicked problem since all the separate sub-systems influence each other in ways often not foreseen upfront.

One way to increase the confidence of correctness in such a system is to apply *Formal Methods* such as the B-method, VDM or mCRL2. Applying a formal method requires to specify the intended behavior of a system in some precise and formal notation. In turn, these specifications can be used to reason about properties of the system (such as safety properties) using theorem proving or model checking. However, the drawback that is often mentioned on the use of such formal method in industry scale projects is that the cost of use (both in time and expertise) is conceived as too high.

A variation on the use of formal methods is to make use of a so called *lightweight formal method*. A lightweight formal method builds on the same principals as the earlier mentioned formal methods but with emphasis on partiality. This can either be partiality in modeling (e.g., model the core design instead of the whole system), partiality in analysis (e.g., perform model checking on a subdomain of the problem), partiality in language (e.g., prohibit a specification language to those constructs which allow for automatic reasoning) and partiality of composition (e.g., allow for the composition of specifications focusing on different aspects for a single system). This emphasis on partiality offers trade-offs to influence the breath and depth of the applied specification and verification technique. In this thesis we explore some of these trade-offs in the context of developing and maintaining enterprise software systems.

Firstly, we explore the impact of supporting automatic verification of a specification has on the design of such a specification language. A very expressive specification language allows for the definition of a large class of problems but will be very hard to automatically verify. We experiment with two different designs in which we balance this trade-off.

Secondly, we focus on partiality of analysis by contributing to the state-of-the-art in relational model finding. Relational model finding is a technique which allows for the definition of problems using a specification language build on a rich relational logic. In existing work of Torlak et al. such a problem is automatically translated into a boolean satisfiability problem which in turn can be solved by an off-the-shelf SAT solver. Our work generalizes this idea by extending the relational input language

to contain definitions and constraints of non-relational data types (such as integers) and by translating this to a Satisfiability Modulo Theories (SMT) problem which in turn can be solved by an off-the-shelf SMT solver. This generalization increases the expressiveness of the relational specification language while still preserving the ability for automatic verification.

Thirdly, we explore partiality of modeling and composition by designing and implementing a specification language and verification method that allows users to perform bespoke specification compositions that allow for partial model checking. These bespoke compositions allow for a modeling technique in which it is possible to specify a complete system while being able to verify parts of it. To offer these partial verification technique we draw upon well known concepts from software testing, *mocking*. To facilitate the verification of these partial system specifications, the specifications are translated to the language of our generalized relational model finder. We apply this new specification technique to a case study from our problem domain, banking systems, and we find that it is expressive enough to specify such a real-world problem while still retaining the possibility to perform checking of user defined properties on a subset of the specifications.

We conclude that the use of lightweight specification and verification techniques can hold value for domains that are currently not quick to adept formal methods.

Titles in the IPA Dissertation Series since 2020

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views.* Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction.* Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming.* Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models.* Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems.* Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing.* Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations.* Faculty of Mathematics and Computer Science, TU/e. 2020-09

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security.* Fac-

ulty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic.* Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components.* Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management.* Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs.* Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and*

Physical Safety are Critical. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions.* Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition.* Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra.* Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing.* Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving.* Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.* Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10

