Fine-Grained Model Slicing for Faster Verification

Author:
Renate Eilers
3701441
r.r.eilers@students.uu.nl

First supervisor:
dr. S.W.B. Prasetya
Supervisor ING:
J. Bosman
Second reader:
dr. J. Hage

Utrecht University

Department of Information and Computing Sciences

A thesis submitted in partial fulfilment (25 ECTS) of
the degree of Master of Science

May 29, 2018

Abstract

In this age of automatization and digitization a majority of organizations relies on large, complex software systems. For many of these organizations, it is critical that their systems behave as expected, as unexpected behavior may result in financial loss, lawsuits, or even human casualties. With software becoming ubiquitous, software validation is growing increasingly important. But software verification is costly, and the resources required to thoroughly check systems are not always available.

We propose model slicing as a technique for increasing the efficiency of bounded model checking, a common method for software verification. Model slicing has been successfully applied to speeding up explicit and symbolic model checking, but the question of whether it will benefit bounded model checking, commonly implemented with highly optimized, state-of-the-art SMT-solvers, is currently unanswered. We provide a model slicing algorithm more fine-grained than the ones found in today's literature, and implement it into a mode slicing tool for Rebel, a modeling language with built-in validation engine based on bounded model checking.

To test our hypothesis we have benchmarked our tool against unsliced models. These benchmarks show that model slicing has the potential to let us check larger problem instances and use higher path bounds. For small and shallow problem instances, however, unsliced bounded model checking outperforms our tool on account of the overhead of slicing. We conclude that bounded model checking can gain a lot from model slicing, depending on model properties such as size, modularity and expected bug depth.

Contents

List of figures

List of Algorithms

List of tables

Acknowledgements

I would like to thank my supervisor and advisor Wishnu Prasetya, who guided me throughout the process of researching and writing this thesis. During numerous meetings Wishnu helped me by proposing literature, asking critical questions, sharing his experience in research and letting me pick his brain when I encountered problems. Another person without whom this work would not have been possible is Joost Bosman, my supervisor at ING who introduced me to Rebel and allowed me to use the bank's resources for this research. Additionally, I would like to thank Jurriaan Hage for lending his expertise in the area of program analysis on multiple occasions.

In the course of this work I have been helped tremendously by Jouke Stoel and Jurgen Vinju, the researchers at CWI who developed Rebel. They provided answers to all my Rebel-related questions and offered useful suggestions for my research.

Finally, I would like to thank Jorryt Dijkstra and Kevin van de Vlist for helping me set up my system to work with Rebel and for showing me the ropes at ING.

1 Introduction

In this age of automation and digitalization, the majority of organizations relies on large and complex software systems for their everyday business. For many of these organizations it is critical that their software systems behave as expected, since unpredicted behavior can result in financial loss, lawsuits, or even worse[Boe72].

Efforts in the area of software verification have resulted in techniques for proving systems free from a range of undesirable properties[Bey14], but constructing such proofs by hand is arduous and error-prone work[DKW08]. With software systems continually growing larger, automatic verification is the preferable—if not the only—option.

Automatic program verification, however, requires considerable computational power and time[CGJ+01]. These are resources that a large number of businesses can not afford to spend. This limitation renders the question of how to make automatic program verification more efficient highly relevant.

In this work we propose a possible solution to this question. We pose that reducing the size of a model, without changing its semantics regarding the property under inspection, may significantly improve model checking efficiency. Model slicing is a technique to extract a (smaller) sub-model from an original model, such that the sliced model contains all parts of the original that are relevant to a given criterion[Sin13].

We test the success of our proposed solution through a case study. For this we will use Rebel[1], a domain specific language (DSL) for specifying systems encountered in the financial world. Rebel was designed by CWI in collaboration with and for the purpose of ING Bank[SSVB16]. The language allows users to specify behavioral properties, and tests whether a given set of programs adheres these properties by employing model checking. To this end, the program specifications and the given properties are translated into a set of logic formulas (SMT), which are then checked for satisfiability by the SMT solver Z3. Rebel will be further introduced in section 3.

The idea of using model slicing to improve model checking efficiency is not entirely new, and previous efforts have been shown to successfully reduce the state space[dL01]. Our contributions are in answering the two following, previously unaddressed questions:

Can we make model slicing more fine–grained? The slicing algorithms described in previous work take as slicing criteria states and transitions in the given model. The algorithms find all other states and transitions that these criteria depend upon directly and indirectly. We ask ourselves whether it is possible to formulate a more fine-grained algorithm, that doesn't just look at the dependences between states and transitions, but at the dependences between specific variables in different parts of states and transitions. In theory, such a variable-specific algorithm could produce smaller slices.

Can model slicing outperform optimizations as applied by state-of-the-art SMT–solvers such as Z3? To improve their efficiency modern day SMT-solvers such as Z3 are equipped with numerous optimizations and heuristics[dMB08]. Currently, there is no literature describing under which circumstances these optimizations are applied. It may be the case that Z3 is equipped with techniques that allow it to exclude irrelevant formulas from its input in a way that is similar to slicing. This means that we do not know whether the cost of slicing will outweigh the cost of checking the full model. By benchmarking our slicing algorithm, we hope to find an answer to this question.

By introducing and implementing a variable-specific, fine-grained slicing algorithm for Rebel specifications and benchmarking our implementation against Rebel's original verification tool, we hope to answer the above questions.

We design our tool to work on well-formed Rebel programs. Thus, it takes into account only the limited type of concurrency encountered in this language (see section 3 for definitions). To extend the algorithm to allow full blown concurrency, the algorithm will need to be slightly altered and extended with additional definitions for the induced dependences.

It should be noted that we omit a formal correctness proof of the slicing algorithm from this work, but we can rest assured in the promising results of empirical evaluations.

Another note on the scope of this research is concerned with the scarce availability of realistic test programs. At the time of this research only a very limited number of real world entities are modeled in Rebel. Our benchmarking thus includes only (extensions of) a small number of existing specifications and some artificial examples.

1.1 Thesis overview

The remainder of this work is structured as follows: first, in section 2 we give an overview of previous work done in the area of optimizing program verification. We review literature on the various approaches to model checking, optimizing SMT-solvers for model checking, program slicing, on

[1] https://github.com/cwi-swat/rebel

which model slicing is based, and finally, we review the main bodies of work done in the field of model slicing.

Section 3 introduces the Rebel specification language. It tells us what specifications may look like, and how Rebel programs are verified.

The next section, section 4, shows all we need to know to slice Rebel programs. It contains the required definitions and algorithms to support slicing, as well as the slicing algorithm itself.

The implementation of Rebel and our slicing algorithm is detailed in section 5. Here, we discuss Rascal, the language in which the both have been implemented, and the data types we have used to represent the formal definitions introduced in the previous section.

Section 6 explains how we benchmarked our implementation. It shows the specifications used for benchmarking and explains why these specifications were chosen. It shows the results of benchmarking, discusses results and mentions some additional findings that may be of interest.

In section 7 we use the results from the previous section to answer the two research questions stated above.

Finally, section 8 lists some of the still open issues encountered during this research.

2 Related work

This section is dedicated to giving an overview of findings in literature from the field of optimizing model checking.

2.1 Approaches to model checking

Model checking is a way to check if a state-based system model adheres to a given property. Model checking verifies whether a property holds by exhaustively searching a model's entire state-space. Generally, model checking algorithms check if the given property holds for the model's initial state, then for all states reachable from the initial state, and so forth until no new states are reachable or a counterexample to the property is found.

According to a 2008 survey of automatic verification methods[DKW08], model checking allows the verification of complex, temporal logic properties, but it is burdened by what is known as the *state-space explosion problem*: the state-space of a program is exponential in the number of variables and the size of these variable's types.

The survey distinguishes between two categories of representations of models for model checking: *explicit* and *symbolic*. The first is implemented by widely known model checking tools like SPIN[Hol97] and CMC[MPC+02]. Tools implementing this approach store model states directly and rely on graph-based algorithms for exploring the state space. Explicit model checking suffers greatly from the state-space explosion problem because it has to store all visited states to keep track of its progress. Symbolic model checking was introduced as a solution to this problem. It relies on implicit representations of sets of states. A successful, commonly used tool implementing this method is NuSMS[CCGR99]. Symbolic model checking has been shown to scale very well in comparison to explicit model checking, verifying systems with up to 10^{20} states compared to a few thousand[BCM+90].

In Rebel specifications, business entities are modeled by extended finite state machines. The translation from these machines to those used in explicit model checking—systems of states and transitions—- seems natural. However, since one of Rebel's aims is to favor efficiency when it comes to verification (see section 3), symbolic model checking, which is said to be sound, complete and efficient[DKW08], may be the better candidate.

The next technique described in the survey is bounded model checking; the approach to model checking currently implement by Rebel.

Where explicit and symbolic model checking verify properties for full models, bounded model checking looks only at finite paths with a length up to a given bound. Paths longer than this bound are not considered. This makes bounded model checking efficient, provided any counterexamples are shallow, but it is not complete for programs containing deeper loops.

Applying what we've learned from the above survey to Rebel, which needs to balance efficiency and safety, symbolic model checking and bounded model checking each have their appeal. In the following we will take a closer look at literature on both.

For symbolic model checking, we consult *Linear Temporal Logic Symbolic Model Checking* by K. Rozier[Roz11]. This 41-page survey provides an overview of the history of symbolic model checking up to the state of the art at the time of its writing (2010).

The introduction of this work states that "Once the system model and specification have been determined, the performance of the model checking step is often very fast, frequently completing within minutes." The author then introduces the state explosion problem, stating that a system with n variables ranging over a domain of k values requires at least k^n states in general. For models using real values the state space thus becomes infinite. For these systems, the author recommends using verification techniques providing weaker assurances or alternative techniques, such as theorem proving. Bounded model checking is also mentioned as a method for reasoning about models whose state spaces exceed the capacity of symbolic model checking.

Some of the statements above appear problematic in the context of Rebel. Assuming that all processes relevant to the banking domain concern values that can be modeled by 32-bit integers rather than real numbers, we are looking at $2^{32 \cdot n}$ states, with n being the number of system variables. A model containing over 3 integer variables would already exceed 10^{20} states, the upper bound of that was mentioned as a breakthrough for state-of-the-art model checkers earlier[DKW08]. This upper bound was said to be solved in 22 minutes[BCM+90], but it should be noted that this benchmark stems from 1992. Modern processors would of course be quicker to return a result. Still, it raises the question of how Rebel programs can be modeled using smaller variable types without losing correctness.

The next section of the survey is concerned with specifying behavioral properties. *Linear temporal logic* (LTL) and *computation tree logic* (CTL)[2] are introduced as logics for expressing desired program properties in. LTL reasons about linear traces through time, whereas CTL reasons about many traces at once. Their expressiveness is incomparable. To make the difference between the two clear, the author states that intuitively, LTL is not capable of describing situations where distinct behaviors occur on distinct branches, while CTL can not express the same behavior happening on distinct branches at distinct times. The author notes that, theory aside, LTL turns out to be generally more expressive in the practice of model checking.

Both logics rely on different model checking algorithms with different running times. If we let $\|M\|$ be the size of the model's state space and $\|\phi\|$ that of the formula representing the behavioral property in total number of symbols (this includes prepositions, logical connectives and temporal operators), then current algorithms for CTL run in $O(\|M\| \cdot \|\phi\|)$, while algorithms for LTL require $\|M\| \cdot 2^{O(\|\phi\|)}$ time. The author notes that this comparison is misleading, as CTL formulas tend to be longer and more complicated. They state that in practice the difference is slight enough not to make computation time a factor in the decision of which logic to use.

As we will explain in more detail in section 3, in Rebel behavioral properties are described by a configuration of multiple entities, possibly with restrictions on their variables and state. The question that pops up after reading about LTL and CTL is: is there a sensible translation between these state configurations and CTL/LTL formulas? And if not, is it desirable to change the way program behavior is specified in Rebel to one that uses LTL/CTL formulas? Rebel was designed to be understandable for product owners and programmers alike. Hence, the answer to the last question is no.

The next section of the work under consideration tells us that the combined system of model and formula can be represented efficiently using binary decision diagrams (BDDs). The author states that choosing a Boolean encoding for an automaton is "a bit of an art", and suggests a method where a binary encoding for every state in the system is followed by a binary encoding of every variable assignment of the system. For instance, if a system consists of four states and two boolean variables, the tuple (1,0,0,1) would represent the situation where the the model is in the third state, one of the variables is false and the other is true. Non-binary variables are encoded using longer sequences. Transitions between states can be modeled by pairs of states: for every bit i in the binary representation of the start state a corresponding variable σ_i is created, along with a variable σ'_i representing the value of the same bit in the resulting state. It should be noted that the size of the resulting BDD depends highly on the ordering of the variables. Finding the most efficient ordering is an NP-complete problem, and in the worst case BDDs can grow exponential in size with the number of variables. A BDD representing an automaton and another one representing the negation of the property formula can be combined into a single new BDD, in which every satisfiable path constitutes a counterexample.

The final section of the survey currently being discussed summarizes what was previously stated, and tells us that scalability is still the biggest issue with symbolic model checking. The author suggests that bounded model checking, though not providing the guarantee that no counterexample of any length exists, vastly increases the size of systems that can be verified, saying that it provides termination within a reasonable time frame.

Considering the author's final remarks, we change the subject of our focus to bounded model checking. For an introduction to the field, we look at *Bounded Model Checking* by Biere et al[BCC+03]. This work begins by motivating the necessity of bounded model checking, stating that "full verification of many designs is still beyond the capacity of BDD based symbolic model checkers." Although bounded model checking requires exponential running time, the authors mention experimental results showing that BMC is capable of solving many cases beyond the scope of BDD-based techniques.

The work then introduces the basic idea of bounded model checking. Rather than translating computer programs into SMT satisfiability problems, as is Rebel's approach, the authors reduce bounded model checking to the boolean satisfiabil-

[2]For a solid, thorough introduction into LTL and CTL, see [BK08].

ity problem. They describe the semantics of model checking under bounded traces, and what it means for a formula to be valid in bounded paths. Of special interest here is the LTL operator **G**, which specifies that some property should hold globally. As bounded model checking only checks paths up to a bound k, paths which are not infinite loops, can never model such a property. This removes the duality between the operators **G** and **F**, which signifies that some property will eventually be true. The authors supply a theorem, stating that the semantics of bounded and unbounded existential model checking (that is, looking for the existence of a path satisfying some property) are equivalent as long as a high enough bound is considered. This make us wonder about the existence of heuristics for selecting the right bounds.

The next section of the work under consideration is concerned with the problem of reducing bounded model checking to propositional satisfiability. The scheme for translating models into formulas described here is very similar to that of Rebel's, as we will see in section 3. It yields an $O(|f| \cdot k \cdot |M|)$-sized formula, where $|M|$ represents the size of the description of the initial state and the transition relation. It should be noted that, using SAT solving and propositional logic, we run into the problem of greatly increasing the number of variables when using data types other than booleans, as we saw earlier in [Roz11]. These variables are needed to represent non-binary data types, and can potentially lead to an exponential explosion of the formula's size.

In the works on model checking that we discussed above ([Roz11; DKW08]), we saw that the lack of completeness is one of the major downsides of using bounded model checking. The final section of *Bounded Model Checking* is dedicated to ways of accomplishing correctness for this technique. The authors describe three methods.

The first is aimed at cases where the property does not contain nested operators. For every finite state system M and property p, there exists a number CT, the completeness threshold, such that the absence of errors up to cycle CT proves that $M \models p$. Worst case, this number is $O(d^{|V|})$, where d is the size of the domain for the variable with the largest domain in $|V|$, V being the set of variables in the model. If the completeness threshold is lower, heuristics can help finding it sooner.

The second technique the authors mention is aimed at liveness properties, which Rebel test configuration can not specify currently. Hence, we skip over this technique. The third method relies on manually finding an inductive invariant implying the property that is being verified. As Rebel verification should be fully automated we are not discussing the method any further.

The article's section on experimental results shows that in the great majority of all benchmarks, BMC-based model checking using SAT-solvers proved significantly faster than the BDD-based algorithm it was compared to. This holds especially when the SAT-solvers were tuned specifically for BMC.

The results of the above suggest that bounded model checking is the favorable technique when efficiency is paramount. For this reason, the next two sections are dedicated to methods for making bounded model checking more efficient. The first looks into the current body of work in tweaking solvers for BMC, and the second lists ways to reduce model size for faster verification.

2.2 Optimizing BMC I: Tweaking SMT solvers

In this section we consider approaches to optimizing bounded model checking that apply heuristics on the solver-side.

The first work we look at is called *Tuning SAT Checkers for Bounded Model Checking* by Ofer Shtrichman[Sht00]. It describes four methods of improving the efficiency of BMC by using the characteristics of BMC formulas. We will describe the four techniques below.

Constraint replication is the first method the authors introduce. This technique is based on the idea that the formula synthesized to describe the model is almost symmetrical. If certain variable assignments always produce at least one unsatisfiable clause, they can be used to to create what the author calls conflict clauses. Imagine, for instance that assigning 1 to x_2 and 0 to y_4, where x_2 refers to the value of x in the second step, will always yield an unsatisfiable clause. This gives rise to the conflict clause $\neg x_2 \vee y_4$. The author claims that these conflicts must hold for any step because of the symmetric nature of BMC formulas. Hence, a *replicated clause* which is a generalization of the conflict clause, is added to narrow down the search space. In the case of our running example the replicated clause would be $\neg x_i \vee y_{i+2}$.

The next technique the author discusses is static ordering. The main idea here is that the variable ordering in which possible assignments are tried can affect the amount of backtracking that must be done later on. The author proposes several options for ordering the variables. One is to apply a breadth-first search backwards on the dependency graph of the program, starting from the property values up to the initial state. As a result, conflicts will be solved more locally.

The next method is similar to the previous one, and consists of combining dynamic and static variable ordering. The author proposes that the first m variables are chosen statically, and the remain-

der is chosen dynamically.

The final technique we see is concerned with choosing the next branch in the search tree. Static ordering does not tell which value to give to a selected variable. The author proposes four options for choosing a value. The first is dynamic, and consists of choosing the assignment that would satisfy the largest number of clauses. The second option is to always choose a constant or random value. The third is concerned with searching for a flat counterexample: in practice variables are unlikely to swap values more than two or three times in a computation. So if the variable under consideration has already been assigned a value in a previous cycle, assign it the same value. The fourth and final option is to repeat the previous assignments: when backtracking from level n to level m, simply keep all assignments between $m+1$ and n the same.

To evaluate the described techniques, the author has benchmarked all possible combinations on 13 designs which were previously proven false. The results were compared to `RuleBase`'s benchmarks on the same problems. `RuleBase` is IBM's BDD-based model checker which, according to the author, is one of the strongest verification tools on the market. The evaluation showed that the static order strategy had the strongest impact on the solver's efficiency. In all cases, it performed significantly better than normal SAT solving as done by *Grasp*. In most cases, constant variable assignment proved most efficient, which the authors explained by the other methods' computational overhead. In only three of the thirteen cases, `RuleBase` outperformed bounded model checking.

Static variable ordering turns out to be only method mentioned in the above work to significantly improve SAT solving. Unfortunately, Z3, the SMT-solver that Rebel relies on, is hard-coded to select unassigned variables randomly[3]. An option would be to fork our own implementation and adding the functionality for users to control the order of variable selection. But now let us consider alternative ways to gain efficiency on the SMT solver's side.

Next, we look at the effects of compiling bounded model checking formulas into SMT rather than propositional formulas, as used to be the standard. We look at *Bounded Model Checking of Software using SMT Solvers instead of SAT Solvers* by Armando et al.[AMP06] The decision to work with SMT rather than SAT is motivated by the fact that SMT formulas can be far more compact than their propositional counterparts.

The introduction gives a quick introduction to SMT. Given a decidable theory $\mathcal{T}$, such as linear arithmetics or the theory of arrays, and a quantifier-free formula ϕ, an SMT solver can determine whether there is a $\mathcal{T}$ model for ϕ.

The work describes an algorithm for transforming programs in a basic, imperative language into SMT. The first phase of this process consists of preprocessing. Loops are unwound into nested `if`-statements, non-recursive functions are inlined and the program is transformed to if normal form: all `else`-statements are removed and `if` statements are moved inwards.

The next phase consists of encoding the resulting program into SMT formulas, which are solved in the next phase. When using SAT solvers, basic data types are converted into bit-vectors. The advantage of using SMT is that the resulting formula is not influenced by the size of the underlying data types.

The fourth and final step consists of building an error trace. The paper uses the SMT solver CVC Lite, which returns the set of formulas rendering the synthesized formula unsatisfiable. The program at hand is then traversed, beginning from the first statement. Traversing the program is straightforward: as it is in `if` normal form, the algorithm simply has to check whether the returned set satisfies the guard at decision points. All encountered statements are collected. When a violated assertion is encountered, the sequence of statements leading up to it is printed.

The authors developed a prototype implementation of the method described above, and have used it to test effectiveness. They benchmarked this prototype on several programs including bubble sort, selection sort and Prim's algorithm for finding a minimum spanning tree for connected, weighted graphs. They compared the results to those of a SAT based method for the same problems. For all the benchmarked programs the SMT method was either slightly slower or slightly faster than the SAT-based one. It did, however, turn out that the SMT based method can handle much larger instances of all problems, as the formulas representing the SAT-based approach grow an order of magnitude faster than the SMT based method, rendering the system on which it runs out of memory.

Our main take-away from the above work is that it is unlikely that switching from SMT-based model checking to a SAT-based method will improve the model checker's speed. We look at other possible ways to improve the efficiency of SMT-based verification.

In the work on bounded model checking that was discussed above([BCC+03]), we saw that SAT-based BMC preforms especially well in cases where the solvers were tuned specifically for using BMC formulas. Rebel's current verification engine is built on the SMT-solver Z3. To see how Z3 com-

[3]This information was extracted from Z3's source code, see https://github.com/Z3Prover/z3

pares to other SMT solvers, we consult last year's International Satisfiability Modulo Theories Competition (SMT-COMP 2016). This competition has been held annually since 2005 to compare state of the art SMT solvers. From the 2016 benchmarks[CDHW] the solvers Z3 and CVC4 stand out. Both outperform all other benchmarked solvers by far. For the competition's application track, which focuses on evaluating solvers interacting with an external framework for the purpose of model checking Z3 is a clear winner. Like Z3, CVC4 does not support user provided heuristics for variable selection[4]. Considering that model checking is the reason we are interested in SMT solvers in the first place, Z3 appears to remain the best choice. But are there other techniques that could improve Z3's efficiency?

The 2013 paper *The Strategy Challenge in SMT Solving* by Leonardo de Moura et al.[dMP13] introduces the notion of strategies in SMT solving. Although the authors state that it is difficult to formally specify what the term means, they describe strategies as "*adaptations of general search mechanisms which reduce the search space by tailoring its exploration to a particular class of problems*". The authors state that although all SMT solvers have built-in strategies, they know of none which have theirs documented. Some of the recent SMT solvers do have parameters allowing users to have some control over the solver's behavior. Z3 is one of these solvers. The authors state that the current version of Z3 allows 284 of such parameters.

The authors introduce the notion of tactics. Tactics describe big symbolic reasoning steps, and can be composed into strategies with tactic combinators called tacticals. Tactics are formally described as functions working sequences of formulas.

Some of Z3's built-in tactics include simplification and converting formulas to conjunctive normal form. Its tacticals include the then-combinator, which takes two tactics, applies the first on the goal and the second on the resulting subgoals. The or-else-combinator takes two tactics, applies the first. If the result is failure, it applies the second. Several tacticals for parallel application exist.

To demonstrate the benefits of using strategies, the authors have benchmarked different strategies to a number of different types of program families for quantifier free linear integer arithmetic. These evaluations show how using certain strategies can significantly improve the time needed to solve a problem. Other strategies, however, can slow the process down a lot— although they may fail less frequently.

The above work shows that applying the correct strategy can be used to improve model checking efficiency in certain cases. However, it is not directly clear what kind of strategies would be beneficial in the case of formulas for bounded model checking. This would certainly pose an interesting research question. For now, we look at a different approach to speeding up BMC.

2.3 Optimizing BMC II: Reducing model size

In this section, we turn ourselves to potential optimizations on the model side. Since the time required for model checking depends largely on the model's size, a rather obvious idea would be to restrict the model somehow. Fortunately, this idea is not entirely new. A technique called *model slicing* allows users to distill from a model only the parts relevant to a given criterion. Model slicing stems from the notion of *program slicing*, which applies the same idea to programs rather than models. To better understand this technique, we turn to *A Vocabulary of Program Slicing-Based Techniques* by Josep Silva[Sil12]. This work is a recent (2012) survey of program slicing techniques. For every distinct flavor of program slicing (there are many), it specifies what the semantics and mentions main applications. But first, a small history:

The notion of slicing was introduced in 1981 by Mark Weiser[Wei81] to help students understand and debug computer programs. Weiser described slicing as is a formalization of the process that programmers go through mentally when debugging their code. A program slice consists of a subset of statements in the original program, and preserves its the behavior with respect to the variables that have been marked as interesting. Weiser defined slices as executable programs consisting of a subset of the original. A slice is based on a slicing criterion, which is a pair (s, v) where s is a statement of the original program, and v a set of variables. Some modern techniques do not demand that the produced slice is executable, as this is not necessarily interesting for all applications.

The original definition of program slicing as given by Weiser is now called static slicing. Static slicing makes no assumptions about the program's input, and thus the resulting slice should contain all parts relevant to any possible execution. To compute a slice, we must know how all program statements depend on each other. A program's *Control Flow Graph* (CFG) makes these dependences explicit. However, CFGs store only control dependences and not data dependences. Thus, to fully specify the relations between statements graphs must be annotated with data flow information as well. Combining these types of information into a single graph yield *Program Dependence Graphs* (PDGs), which have proven optimal for program slicing as it allows users to build slices in

[4]This also became clear from inspecting the source code at https://github.com/CVC4/CVC4

linear time with respect to the number of nodes in the PDG. The cost of building a PDG is quadratic in the size of the program.

We now briefly list the various slicing techniques, described in Silva's survey, that may be interesting to our application.

The first technique the survey touches on is as the original described by Weiser. Today, it is called static slicing. Static slicing tells us which program statements can possibly influence a given set of variables at a given program point. Its main applications are testing, program comprehension and dead code removal.

Next is dynamic slicing. In contrast to static slicing, which accounts for all possible inputs, dynamic slicing is used when one is interested in a particular execution. Dynamic slices are smaller than static slices, because the knowledge of input values rules out certain program behaviors. Slicing criteria for dynamic slicing are like those for static slicing extended with input values. Dynamic slicing is mainly used for debugging and testing.

Slicing can occur in two directions: forwards and backwards. Backwards slicing is used when one is interested in all statements that could influence the slicing criterion, whereas forward slicing tells us which statements are influenced by a given criterion. That is, it says what other parts of a program will be affected by a given modification in one part. Forward slicing is mainly used for dead code removal and software maintenance.

A generalization of forward and backward slicing is the technique dubbed 'chopping'. In chopping, the slicing criterion consists of two sets of variables: the source and the sink. A slice resulting from chopping consists of all statements being affected by the source and affecting the sink. Chopping with an empty source results in backwards slicing, whereas an empty sink has the same result as forwards slicing Chopping is used for program analysis and debugging.

Another technique Silva describes is called hybrid slicing. As the name suggests, it is a combination of dynamic and static slicing. Dynamic slicing is more precise than static slicing, but it is also more computationally expensive. Hybrid slicing increases the precision of static slicing by adding dynamic information into a static analysis. This is done through a set of breakpoints, where information about which program parts have been executed up to the breakpoint can be entered. For the set of executions defined by these breakpoints, hybrid slicing tells a user which statements could influence the slicing criterion. Its primary application is debugging.

Slicing as originally defined by Weiser did not take procedure calls into account, and upon later research PDGs proved insufficient for representing multi-procedural programs. Thus, when a program contains procedure calls, some precision is lost in traditional slicing. To combat this loss of precision, interprocedural slicing was introduced. This technique computes on a *System Dependence Graph* (SDG). Like PDGs, SDGs contain control and flow relations. Additionally, they contains interprocedural flow relations and parameter relations. Later, it turned out that using PDGs together with call graphs results in the same level of precision.

The next technique on the list is quasi-static slicing. This method produces a slice with respect to a particular set of executions. It can be used when a set of input values are fixed, and the rest is unknown. It answers the question: for a set of executions where some inputs have known values, what statements can influence the slicing criterion? It's used for debugging and program comprehension.

Next, we consider call-mark slicing. This technique reduces the cost of dynamic slicing by reducing precision. The slices that call-mark slicing produces are smaller than static slices, but bigger than dynamic ones. The technique uses dynamic information when constructing the PDG: it marks actually executed call statements, and removes unmarked ones from the PDG. The graph can then be traversed using standard static techniques.

Dependence-cache slicing is another technique based on pruning the PDG with dynamic information, but instead of removing nodes, as call-mark slicing would, it removes edges. On average, the technique is more precise than call-mark slices, and also less expensive to compute.

Dicing is another slicing technique. It produces the difference between two slices for different variable sets, usually where one shows desirable behavior, and one undesirable. This can be useful in the context of debugging.

The technique of barrier slicing relies on user input. It allows users to specify which parts can be traversed and which can not during the construction of the slice. It is mainly used for program comprehension.

Another potentially interesting technique is conditioned slicing. Like quasi-static slicing, conditioned slicing computes slices with respect to a set of initial states. But instead of describing these states by a partial input, the user must supply a first-order logic formula. The criterion is a quadruple with a subset of input values, a logic formula on these variables, a statement and a set of variables of interest. It is mainly used in debugging and program comprehension.

Conditioned slicing can also be done in a backwards direction. This places the condition anywhere in the program, and looks at the statements required to make it true. This answers the ques-

tion of how the program could get into the state described by the formula. A widely used application for backwards conditioned slicing is program specialization and program comprehension.

A generalization of both forward an backward condition slicing is pre-/postconditioned slicing. This technique removes all statements except those which can be executed according to the precondition, and which can satisfy the negation of the postcondition. It is used for program comprehension and verification.

A very interesting slicing technique, because it differs in essence from all the ones we have seen up to this point, is amorphous slicing. Where all techniques we have encountered which preserve both the semantics and the syntax of the original program, Amorphous slicing makes do without this last condition: it may apply program transformations in the process. This allows for greater simplifications, resulting in smaller slices. It answers the question: can this program be changed to only compute a set of given variables at this program point? It is used for program comprehension.

This brings us to concurrent slicing, which is a technique on its own because concurrency can not be represented by the standard graph representations. However, CFGs and PDGs can be extended with special nodes for representing parallel execution. This introduces the notion of interference. In contrast to control and data dependence, interference is not transitive. This means normal slicing algorithms fall short. Algorithms for concurrent slicing need to take interference relations into account somehow. Various different algorithms for concurrent slicing exist, each dealing with interference's lack of transitivity in its own way.

The final type of slicing that the author introduces is proposition based slicing. This technique was created to reduce finite state transition systems used in model checking. The user specifies a formula, from which the slicing criterion is derived automatically. The slicing criterion contains a pair (s, v) for every statement appearing in the formula, as well as for its predecessors and successors in the CFG. Here, v are the variables appearing in the formula. Additionally, such pairs of statements are created for all statements assigning a value to any variable in v. In contrast to other techniques considered so far, it produces statements not relevant to the final value of the variables in the criterion. It answers the question of what statements are needed to satisfy a given LTL formula.

This, finally, concludes the survey of program slicing. As we saw, there is a wide range of different slicing techniques. Most answer a slight variation of the same question. But how do we choose which suits us the best? For instance, do we need interprocedural slicing, or is the intraprocedural variant sufficient because functions are inlined during verification anyway? Amorphous slicing may produce even smaller slices than the other techniques mentioned above. This makes it potentially it relevant, but it also raises the question of how traces in amorphous slices can be translated back into traces on the original model for debugging. Although we can probably come up with some inverse strategy for this, we have to wonder whether the time such a procedure would require outweighs what can be gained from it. Techniques like conditioned slicing, pre-/postconditioned slicing or proposition based slicing may be interesting too, as they take up some of the model checker's work. But as it is likely that such complex slicing techniques are less computationally efficient, it may be the case that using a simpler algorithm to produce a bigger slice for the highly optimized SMT solver Z3 is actually faster. In general, we will have to find out if the added cost of program slicing does not actually slow the process down on account of Z3 heuristics outperforming it.

Another thing we will have to keep in mind is that models are not sequential computer programs. Thus, the techniques described in the survey are not directly applicable to Rebel code. We will have to look at how dependency and slicing criteria as defined above translate to models, and think of properties we actually demand of a slice. To address these issues, we turn to another work. *State-Based Model Slicing: A Survey* by Androutsopoulos et al.[ACH+13] reviews existing work on slicing state-based models (SMBs).

The work begins by motivating the importance of state-based models in computer science. The authors state that models are capable of conveying certain kinds of information better than programs, and hence the two are not interchangeable. Models are, however, more prone to becoming too large for practical applications. This makes model slicing an essential technique.

As we stated before, program slicing techniques are not directly applicable to models: state-based models are essentially graphs, whereas programs are sequences of statements. Cutting away parts of a graph may results in connectivity problems that do not occur when leaving out part of a sequence. Additionally, program slicing algorithms generally operate on lines of code, but SBM's do not have such a level of granularity: a single node in a SBM may represent several lines of code, and inversely a single statement may be represented by multiple nodes and edges. Thus, where program slicing can simply remove lines of code, SMBs must sometimes perform extra transformations to prevent nodes from being orphaned.

Another concern not addressed in normal program slicing relates to the non-deterministic nature

of SBMs. While non-determinism is commonly encountered in state-based models, most programming languages tend not to allow for it. Dealing with non-determinism in model slicing algorithms is still an open issue.

A section of the survey that is of particular interest to us is one dedicated to various approaches to model slicing specifically for the purpose of model checking. The authors suggest that, in this context, slicing can be done on either the input language solver's input language (SMT-LIB in the case of Rebel), or on the model before it is translated. This section of the survey presents results for the latter, by mentioning promising experimental results for models in UML, EHA and RSML.

But despite this apparent success, the field of state-based model checking is relatively unexplored according to the authors. This is demonstrated by the final section of the survey, where they discuss a number of issues that are still open. One of them is concerned with slicing non-terminating models, which the authors state there are currently no correct ways of accounting for.

Another issue the authors point out concerns precision. This is a real problem for hierarchical state machines in particular. Most algorithms working on this type of model start with the lowest level in the hierarchy and consider all states at that level before moving up. If a state is in a slice, then so are its superstates. But then all substates forming the superstate must be included too, leading to larger slices which are less precise.

Concurrency and communication pose another problem: most current approaches to slicing handle communication by introducing new dependences, which is complex and tends to induce a lack of precision. If the slicing algorithm assumes transitivity of dependences here, it can be incorrect. As we saw in the survey [Sil12], this is an issue for program slicing as well.

Next on the list of issues is graph connectivity. Some algorithms mark elements that should be kept, other algorithms mark elements that should be removed. Most algorithms do not delete states or transitions that can cause other states to become unreachable. This leads to larger, less precise slices. The authors state that only one algorithm exists which removes transitions and reconnects the machine by merging states. Although it is appealing to try and find the smallest possible slice, this brings forth the problem with amorphous slicing that we touched on earlier: as the model will look different, can a resulting counterexample still be used to construct a trace that is useful for debugging the original model?

As a final issue, the authors mention the problem of slicing richer and larger state-based models: only 'simple' features such as hierarchy, concurrency and communication are accounted for currently. This limits the expressiveness of models that can be sliced. More research needs to be done into techniques to account for more features.

The survey gives us some insight on the state of the art in model slicing. A small number of promising results in this area make us believe that it may be possible to construct an algorithm for slicing Rebel code, which may improve the efficiency of model checking. However, the survey also shows us that a number of questions in the area are still unanswered. None of the model types translate to Rebel in a way that is directly clear, which leaves the possibility that the existing algorithms and theories will not be able to account for all of Rebel's features.

In this section we looked at the spectrum of automatic verification techniques currently available. This review showed that full (symbolic) model checking is too inefficient for our purposes. For real world systems, state of the art verification engines need minutes or even hours rather than seconds, which clashes with Rebel's philosophy of providing safety assurances on the fly. Since bounded model checking can significantly improve computation time by limiting the depth of the verification, this method seems best suitable to our needs. But even bounded model checking is costly in terms of time and memory. In the literature, we found three promising techniques for speeding BMC up. They are listed below:

- Tweak the SMT solver to instantiate variables in a static order based on their dependences;
- Steer the SMT solver by using strategies optimized for solving BMC formulas;
- Perform model checking on a smaller model by slicing to the specification before checking.

All of these options have shown some success in speeding up verification. Which should we go with? A factor in our decision is the fact that researchers at CWI are currently looking into methods of making the translation from Rebel specifications to SMT-LIB more efficient by means of an intermediate language. This is likely to have an impact on the structure of the resulting formulas. But as it is a work in progress, not much can be said on what the final BMC formulas will look exactly. Hence, we refrain from focusing on strategies catered to the current SMT-LIB representation at this point in time.

Instead, we look at model slicing. This technique limits bounded model checking to only those parts of the specification which potentially affect whether the property that we are trying to verify holds. This technique has potential for speeding up the process, as in theory verification time depends on the size of the model under inspection. In the worst case, slicing will yield the exact same

model we started with, but this seems unlikely as we know from practice that code tends to express at least some degree of modularity.

3 The Rebel specification language

3.1 Rebel specifications

In this section we give an in-depth description of the Rebel language as specified in [SSVB16]. At times we diverge slightly from this document, as Rebel has seen some (yet unpublished) renovation since the writing of [SSVB16]. Our knowledge of Rebel's recent revisions comes directly from working with Rebel and its developers.

In Rebel, business entities—e.g., bank accounts, transactions—are modeled by state-based machines. Rebel was designed for the domain of finance, and as such it comes with several built-in primitive types such as `Money`, `Currency` and `IBAN`. A Rebel specification of an entity consists of three main parts: a number of fields containing state variables, a set of event declarations and its life cycle: a description of the allowed transitions between states.

Figure 1 shows a specification modeling a bank account. It contains the fields `balance`, holding an instance of the `Money` type, and `accountNumber` which is an IBAN. `accountNumber` is the entity's unique identifier, which is determined by the `@key` extension following its type declaration.

Between line 17 and 27, the `Account`'s life cycle is declared. It defines an initial state called `init` (determined by the keyword `initial`) and a final state `closed`, as determined by the keyword `final`. The life cycle tells us which transitions between states are allowed. Life cycle descriptions are formatted `s1 -> s2 : e1, e2, ..., en`, and should be read as '*if* the machine is in state `s1`, the occurrence of event `e1` or `e2`, or ..., or `en` puts the machine in state the state `s2`'. Only one event can occur in a single entity at any time. Events are treated as atomic. In the example demonstrated in figure 1, the occurrence of an `openAccount` event in the initial state brings us to the `opened` state.

The `Account`'s events are declared between lines 7 and 15. Events can have parameters, and entities can bind these. We see this occurring in line 8 for the `openAccount` event. Unbound parameters either use default values, which can be specified in event definitions, or they obtain their value from other parameterized function calls. Event definitions are stored in dedicated library specifications. This separation of declaration and implementation was introduced to encourage reuse: multiple entities can inherit from a single library specification, and a single entity can import multiple libraries.

Figure 2 shows us a specification of the `openAccount` event. The `event` keyword specifies that what follows is an event. Next are the event's

Figure 1: Rebel specification of an `Account`

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

name, and any optional parameters between square brackets. Behind that we see, between parentheses, any optional event arguments. The `openAccount` event has one parameter of the `Money` type called `minimalDeposit`. It is given the default value of `EUR 0.00`. This value will be used if no parameter is provided. The event's argument is called `initialDeposit` and is of type `Money`.

Events may have pre- and postconditions. An event can only take place if all of its preconditions are met. In our example, the event `openAccount` can only take place if the value of `initialDeposit` is greater than the minimal deposit.

Postconditions describe the effect of an event. In our example, we see `new this.balance == initialDeposit`. The keyword `new` here indicates that the value of a variable will be updated. We also see `this.balance`, which means that the variable `balance` of the entity itself is indicated. This distinction must be made to avoid clashes because, as we will see in a bit, entities can refer to the variables and events of other entities. If the event in our example is executed, it should hold that the `Account`'s balance now has the value of the initial deposit.

In addition to specifying sequential automata, Rebel allows basic concurrency. We see an example of this in figure 3. The specification shown here models a monetary transaction between two bank accounts. In line 5 and 6, we see the state variables `to` and `from`, both of which are of the type `IBAN`. After their type declartion we see the extension `@ref=Account`. This extension, `@ref=...`, followed by an entity name, signifies a reference to a concrete

Figure 2: Rebel specification of the `openAccount` event

```
event openAccount[minimalDeposit: Money =
    EUR 0.00](initialDeposit: Money) {
  preconditions {
    initialDeposit >= minimalDeposit;
  }
  postconditions {
    new this.balance == initialDeposit;
  }
}
```

instance of one such entity. It must be referenced by something of the type that is designated to be its unique identifier.

Figure 3: Rebel specification of a `Transaction`

```
specification Transaction {
  fields {
    id: Integer @key
    amount: Money
    from: IBAN @ref=Account
    to: IBAN @ref=Account
  }

  events {
    start[]
    book[]
    fail[]
  }

  lifeCycle {
    initial uninit -> validated: start
    validated      -> booked: book
                   -> failed: fail
    final booked
    final failed
  }
}
```

When we declare such references to concrete instances we expect all of them to run concurrently. Rebel does not allow the sharing of fields, but a limited form of communication is available. We see an example of this if we take a closer look at the definition of the `Transaction`'s `book` event in figure 4.

Figure 4: Rebel specification of the `book` event

```
event book() {
  sync {
    Account[this.from].withdraw(this.amount
    );
    Account[this.to].deposit(this.amount);
  }
}
```

Other than pre- and postconditions, event definitions allow synchronization blocks to be declared, signified by the **`sync{}`** syntax. Statements within these block reference the events of other entities that are to take place synchronously with the event that is being defined. This means that, during the time step the event containing the **`sync`** statement is takes place, all referenced events should take place, too. Synchronization statements are of the shape `T[id].e(a1,...,an)`, where `T` must be the name of a specified entity, `id` must be a value of the same type as `T`'s unique identifier, `e` must be an existing event of `T`, and `a1` to `an` should be values corresponding in type and number to `e`'s expected arguments.

Lines 2-5 in figure 4 show us a synchronization block occurring in the `book` event of a `Transaction`. Line 3 tells us that booking must be synchronized with a `withdraw` event from the `Account` identified by the **`IBAN`** value bound to the `Transaction`'s variable `from`.

Figure 5 gives a graphical depiction of the way a `Transaction` and its two referenced `Accounts` interact.

3.2 Rebel TestModules

In this section we introduce TestModules, the modules used for describing certain properties of Rebel specifications for verification. We will explain the syntax and semantics of TestModules, show an example and explain the mechanics of the verification process.

If a user wants to check a specification written in Rebel for certain properties, they can do so by describing this behavior in a TestModule. A dedicated language with Rebel-like syntax was designed to this end. In a TestModule, a user can describe an abstract configuration of any number of Rebel entities, along with an integer bound. A TestModule consists of one or more setup statements, where each statement describes the instance of an entity. A setup statement can either mention one of the entity states described in the life cycle, or leave the goal state unspecified. In addition to a possible state reference, setup statements may contain one or more predicates over the entity's variables. When a TestModule is checked, Rebel's verification engine tries to find a path—no longer than the supplied bound—leading to a state satisfying all setup statements described in the test file.

Figure 6 shows us a Rebel TestModule. It describes a configuration called `negativeBalance` in which we see a `Transaction` in the state `booked`, an `Account` in the `opened` state with a `balance` smaller than 0, and another `Account` required to be in the state `opened`. When the **`check`** statement on line 7 is executed, the solver tries to find a path consisting of at most five steps, including the initial state, to a state where the configuration described in `negativeBalance` holds.

3.3 Verifying Rebel

To verify a TestModule, all relevant specifications and configurations must be translated into logic formulas. These are then compiled into SMT-

Figure 5: Graphical representation of a `Transaction`. The dashed lines separate concurrent automata. The left and right automata are `Accounts`. The middle is a `Transition`. The dotted lines represent synchronization between events.

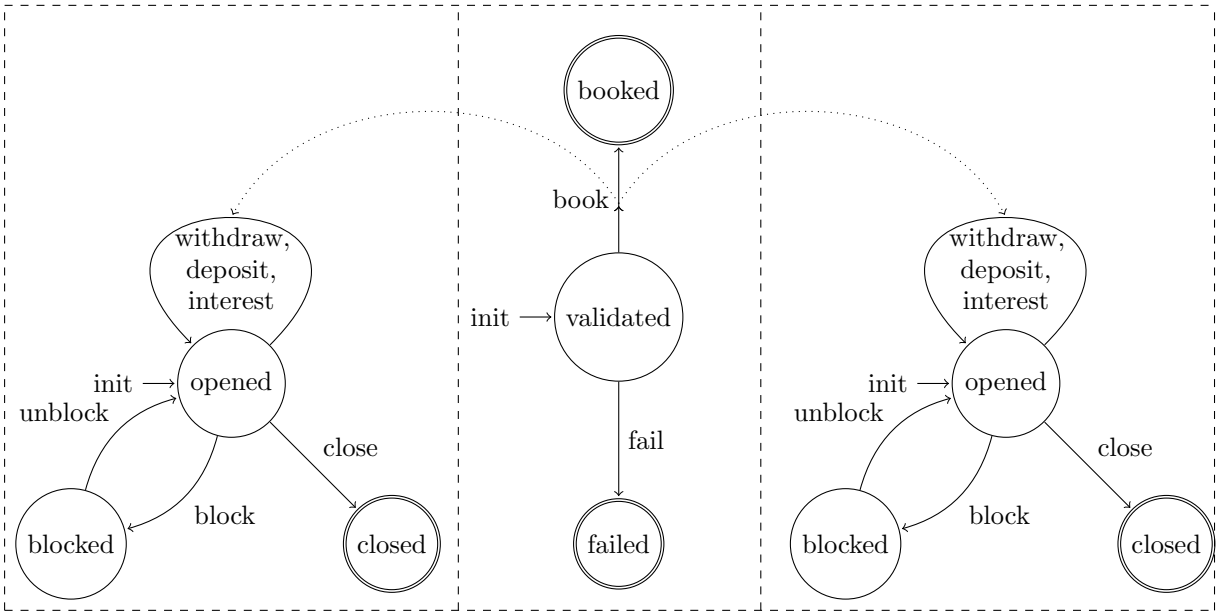


Figure 6: A Rebel TestModule

```
state negativeBalance {
  booked Transaction;
  opened Account with balance < EUR 0.00;
  opened Account;
}

check negativeBalance reachable in max 5
    steps;
```

LIB[BST10], a common input language for SMT-solvers. Th translation into SMT-LIB consists of three steps:

First, a predicate describing the initial state is created. Let's call this predicate I. This predicate states that every entity mentioned in the TestModule is in the state marked as its initial state in its specification. For every instance a unique ID is generated, so multiple instances of the same entity can be distinguished.

For the next step all valid transitions between states are encoded into a formula as a relation on states. Let's call this binary predicate on states T. It is a conjunction of smaller formulas: one for every event e. It must hold that the first state is the source of e, and that all of e's preconditions must hold in this state. The second state must be e's target, where the postcondition must hold. If e contains any synchronization statements, it must hold that all synchronized events take place simultaneously.

Finally, in a way very similar to how I was created, a predicate G describing the goal state is defined. This predicate states that all entities are in the state described in the TestModule, and that any restrictions on their variables declared there should hold.

The three predicates I, T and G are then combined into the single formula below, which is presented to Z3 for a satisfiability check:

$$I(s_0) \wedge \bigvee_{i=0}^{k} \left(\left(\bigwedge_{j=0}^{i-1} T(s_j, s_{j+1}) \right) \wedge G(s_i) \right)$$

This formula describes all paths up to length k, where k is the bound that limits the depth of model checking, such that the first state is an initial state, and the last state is a goal state, and all states in between are reached through valid transitions. If such a path exists, Z3 returns a model for it. From this model, an error trace can be constructed upon request. This allows the user to reconstruct the steps leading up to the undesirable behavior.

This concludes our overview of the Rebel specification language. The next section looks into techniques for slicing programs written in Rebel.

4 Slicing Rebel specifications

Slicing Rebel specifications is a multi-step process. We must obtain a slicing criterion from the TestModule, and we need to analyze all relevant specifications for data- and control-flow to build a graph on which we can run our slicing algorithm, which returns a subset of nodes from the original specification. In this subgraph, nodes may have become orphaned or otherwise disconnected in such a way that the semantics of the original specification are

changed. We restore connection properties, and finally we remove everything that is not in the reconnected slice from the original specifications.

In this section we will walk through all steps of the slicing process in the order introduced above.

4.1 Obtaining a control-flow graph from Rebel specifications

In this section we formalize the notion of a control flow graph for Rebel models, and we explain how to obtain them from Rebel specifications.

A control flow graph (CFG) is a directed graph (N, E) in which the edges signify the control flow paths of a program[All70]. For a node $n \in N$, we let $\pi(n)$ be all direct predecessors of n, that is, all m s.t. $(m, n) \in E$. We let $\sigma(n)$ be all direct successors of n (all m s.t. $(n, m) \in E$). We let $uses(n)$ be the set of all variables that are used at a n's precondition, postcondition or synchronization statement. $defs(n)$ is the set of variables that are defined by either n's arguments or any assignments in n's postconditions. The set of variables V consists of all variables global to the specification—i.e. every entity field—and all local variables—i.e. event parameters or arguments. We define an additional variable ϵ to refer to the execution of a node. This allows us to express control dependences: the effect other program parts may have on whether a node is executed. This idea is based on the work of D. Jackson[JR94], which is aimed at making program slicing variable-specific, and the main inspiration of our variable-specific model slicing algorithm.

The state-based machines that Rebel specifications describe already express every possible flow of execution of an entity, yet we transform them into a CFG to perform slicing. We do this to achieve a more fine-grained level of slicing by breaking events up into separate nodes. This way, irrelevant parts of an event, and all dependences they introduce, can be left out of the final slice. We define four different kinds of nodes: state nodes, guard nodes, assignment nodes and synchronization nodes.

CFGs are obtained from the life cycles of entities as follows: first, we create a state node for every state mentioned in the life cycle. For every event in the life cycle, we create a set of nodes: one 'root' node: a guard node which uses all variables mentioned in any preconditions of the event, and then one assignment node for every assignment in the postcondition, which uses all variables on the right hand side of the assignment and defines all variables on the left hand side. Finally, we define a synchronization node for every synchronization statement, the uses of which are any variables used as parameters in the statement. Together these event nodes form a chain in the graph, linked in the order of execution. For every synchronization node we create an edge to the root node of the events that are to be synchronized. Figure 7 shows us the Rebel specification of an event, and figure 8 shows its corresponding chain of CFG nodes.

Figure 7: An event

```
event e(inc : Integer) {
  preconditions {
      a > b;
  }
  postconditions {
      new this.a == this.a + inc;
      new this.b == this.b + inc;
  }
    sync {
      T[this.ref].e'(this.c);
    }
```

Next, for every entry `s1 -> s2 : e1, e2, ... , en` in an entity's life cycle, we create edges between `s1` and the root node of `e1, ...,` to `en`, and from the final nodes in the chain of nodes corresponding to `e1, ... ,en` to `s2`.

4.2 Dependences

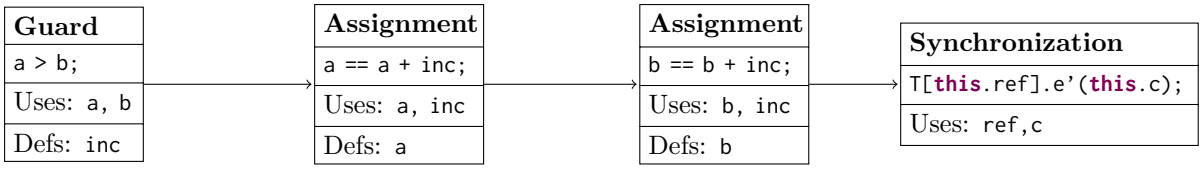
Based on the CFG obtained from a specification, we create yet another graph $G = (N', E')$. We do this to make slicing variable-specific. This new dependence graph relates instances $I \subseteq N \times V$ to instances. The edges in the dependence graph represent dependences between the instances. We distinguish between seven kinds of dependences, the union of which is taken as the set of edges for the dependence graph. Some of these dependences, such as def-use dependence, use-def dependence and control dependence, are familiar from other works on model slicing. Other types are invented specifically to handle dependences induced by the semantics of synchronization in Rebel and the way our CFG's were created. Amongst these are nested control dependence and synchronization dependence. The seven types of dependences that make up the edges of our graph are defined below. It should be noted that, in the following, when we write $p \xrightarrow{d} q$, we mean that p is d-dependent on q.

Use-def dependence, or $\xrightarrow{ud}$, connects the use of a variable at one node to a definition at another node. Formally, if a variable v is used at the instance (n, v), and v is defined at the instance (m, v), *and* there is a *definition-clear path*[5] from m to n, then $(n, v) \xrightarrow{ud} (m, v)$.

Def-use dependence, or $\xrightarrow{du}$, represents internal dependences of a node: it describes how a variable definition at one node depends on uses of

[5] A *definition-clear path* is a path $[n_1, \ldots, n_k]$ from n_1 to n_k, such that n_k uses the variable v, n_1 defines v, and this definition reaches n_k (that is, it is not redefined anywhere on the path between n_1 and n_k. Reaching definitions are calculated by algorithm 1.

Figure 8: The chain of CFG nodes corresponding to the event defined in figure 7



that same node. That is, it tells us which variables were used to calculate the defined variable's new value. If n is an assignment node representing a postcondition statement of the form `new this.v == expr`$(w_1, \ldots, w_i)$, where $expr(w_1, \ldots, w_i)$ denotes some expression over the variables $w_1, \ldots, w_i$, then $(n, v) \xrightarrow{du} (n, w_1), \ldots, (n, v) \xrightarrow{du} (n, w_i)$

Control dependence, or $\xrightarrow{cd}$, connects the execution of a guard node to the uses of variables occurring in any guarding expressions. This signifies that whether the event represented by the node can be executed depends on the values of the guarding variables. Formally, if n is a guard node representing a precondition with a set of expressions ranging over $v_1, \ldots, v_i$, then $(n, \epsilon) \xrightarrow{cd} (n, v_1), \ldots, (n, \epsilon) \xrightarrow{cd} (n, v_i)$.

Nested control dependence, or $\xrightarrow{ncd}$, defines the control-dependence of an assignment or synchronization node to the event's guard node. For any event, if m is the root node of the chain representing it and $n_1, \ldots, n_i$ are its assignment nodes and synchronization nodes, $(n_1, \epsilon) \xrightarrow{ncd} (m, \epsilon), \ldots, (n_i, \epsilon) \xrightarrow{ncd} (m, \epsilon)$.

Def-execution dependence, or $\xrightarrow{de}$, connects the definition of a variable to the execution of the event node in which it is defined: $\forall v \in defs(n), (n, v) \xrightarrow{de} (n, \epsilon)$. It signifies that postconditions take effect only when events actually take place.

Parameter data dependence, or $\xrightarrow{pdd}$, is a particular kind of data dependence. When a synchronization statement in one event forces the synchronized execution of another, the synchronization statement provides parameters for the synchronized event. Thus, the values of the arguments in the synchronized events depend on the values of the parameters in the synchronization statements. This is what parameter data dependence signifies. Let $e(a_1, \ldots, a_i)$ be an event of entity T represented by the node n, and s be a synchronization statement forcing the synchronization of event e of entity T, represented by the node m and supplying e with the parameters $p_1, \ldots, p_i$, then $(n, a_1) \xrightarrow{pdd} (m, p_1), \ldots, (n, a_i) \xrightarrow{pdd} (m, p_i)$.

Synchronization dependence, or $\xrightarrow{sd}$, states that the execution of a synchronization statement depends on the execution of the events that have to be synchronized with it: if these events can not be executed, neither can the synchronization statement itself. If m is the node representing this synchronization statement and n is the guard node representing the event that is to be synchronized with m, then $(m, \epsilon) \xrightarrow{sd} (n, \epsilon)$.

The dependency graph $G' = (N', E')$ is constructed by taking $N' = \{(n, v) \mid n \in N \wedge v \in uses(n) \vee v \in defs(n) \vee v = \epsilon\}$ and $E' = \xrightarrow{ud} \cup \xrightarrow{du} \cup \xrightarrow{cd} \cup \xrightarrow{ncd} \cup \xrightarrow{de} \cup \xrightarrow{pdd} \cup \xrightarrow{sd}$.

4.3 Obtaining slicing criteria from TestModules

In section 2 we saw that, in traditional model slicing, a slicing criterion consists of a set of model states or a set of transitions. As one of our aims is to construct a model slicing algorithm that is more fine-grained, we want our criterion to be more precise: it should be able to take variable-specific dependences into account. To be able to express this, we define our slicing criterion consist of instances (recall that an instance consists of a variable and a state). When an instance (v, s) is in the set of criteria, we are interested in all states and transition affecting v in state s.

Now if we recall from the previous section, every setup statement in a TestModule may specify a goal state and constraints on entity fields. Whether both, only one, or two of these parts are present in a setup statement decides what the slicing criterion will look like. Below we explain how every possible combination shapes the slicing criterion.

Both a goal state and variable constraints are defined. This case is very similar to a traditional program slicing criterion: we are interested in a particular (set of) variable(s) at a particular program point. If a setup statement of this form is encountered, then for every variable v constrained in the setup statement we add the pair (n, v) to the criterion, where n is the node representing the goal state mentioned in the setup statement.

A goal state is defined without any variable constraints. When only a goal state is defined, we must take into account *all* paths

```
begin
    defs_in ⟵ ∅;
    defs_out ⟵ ∅;
    repeat
        foreach n ∈ N do
            defs_in(n) ⟵ ⋃_{m∈π(n)} defs_out(m);
            defs_out(n) ⟵ {(m, v) | (m, v) ∈ defs_in(n) ∧ v ∉ defs(n)};
            defs_out(n) ⟵ defs_out(n) ∪ {(n, v) | v ∈ defs(n)};
        end
    until defs_in and defs_out reach a fixed-point;
end
```

Algorithm 1: Calculating reaching definitions

leading up to this state. We find all states occurring on any path from the initial state to the goal state. For every node representing such a state n, we add the instance (n, ϵ) to the set of criteria.

No goal state is defined, but there are variable constraints. If only variable constraints are defined, any state in which the value of a goal variable can be changed is relevant. Therefore we add all pairs (n, v), where n is the target of a transition which alters the value of the variable v, and v is in the set of constraints.

Goal state nor variable constraints are defined. All paths through the entity should be considered. For all nodes n in the graph, (n, ϵ) is added to the set of criteria.

On some occasions multiple setup statements for one type of entity can occur in a single TestModule. We see an example of this in figure 6, where the goal is to find one `Account` opened, and another one opened *with* a negative balance. In such a scenario, we combine both criteria by taking the union. This approach may not seem ideal: in some cases one setup statement can result in a very small slice, whereas the other requires the full model. It would be preferable to create two separate entities as this would limit the search space for at least one of the entities. Unfortunately, this is not as straightforward as it may seem. Imagine we did do this, and we created a specification `Account1` for the `opened Account` and an `Account2` for the `openend Account with balance < EUR 0.00`. This would be just fine until we consider the `Transaction` entity. This entity references two `Accounts`. Which of these should be `Account1` and which should be `Account2`? There is no way of knowing in advance, and both assignments may have to be explored anyway. We will discuss this some more in section 8, but for the current implementation we simply combine both criteria and create the bigger slice.

4.4 Slicing

Once the dependency graph is constructed and the slicing criterion is known, slicing is straightforward: we calculate all instances relevant to the slicing criterion by traversing the graph starting from the nodes in the criterion. Algorithm 2 shows the slicing algorithm. It is based on the slicing algorithms found in other works on model checking ([Oja07; WDQ02; KLB12]).

Initially the slice is assigned to be the slicing criterion. Every iteration of the algorithm's loop may add new nodes to the slice. When no new nodes are added, the loop is exited and the algorithm terminates. Note that the algorithm will always terminate, as it operates on finite graphs: at one point either no new nodes will be reachable through graph edges from nodes in the slice, and hence the value of *slice* stabilizes, or all nodes in the graph will be in the slice. In this case, no new nodes can be added in the next iteration, causing the slice to stabilize and the algorithm to terminate.

4.5 Restoring reachability relations

When the slicing algorithm finishes we get a set of relevant instances. During the process, nodes that belong in the slice may have become orphaned, or the order in which nodes can be reached may have changed. To maintain the original reachability relations with respect to all nodes in the slice, certain nodes will have to be reconnected.

We have implemented two methods for achieving this. The first, conventional one, adds any missing nodes that were removed by slicing back to the graph until all nodes in the slice can be reached in the same order as in the original. Consider, for instance, the state-based machine in figure 9(a). Assume that $s1$, $s3$, $s4$ and $t4$ end up in the slice, which is depicted in figure 9(b). In the slice we see that $s4$ is no longer reachable from $s1$, even though both are in the slice and can reach each other in the original. Additionally, $s3$ is only reachable by going through $s4$ first, even though it could

```
input  : A dependency graph G' = (N', E') and a set of slicing criteria C
begin
    slice ⟵ C;
    repeat
        foreach i ∈ slice do
            slice ⟵ slice ∪ {j | (i, j) ∈ E'};
        end
    until the value of slice stabilizes;
end
```

Algorithm 2: Slicing

reached before getting to $s4$ in the original. Algorithm 3 describes the first approach to restoring reachability, which traverses and adds parts from the original CFG if they are needed to reconnect disconnected parts of the slice. Figure 10 shows us the result of applying this algorithm to the slice we saw in figure 9: $t1$, $t2$, $s2$ and $t3$ are added. $t1$ and $s2$ are added because the other in-slice nodes can only be reached by going through them in the original graph. $t3$ is added because it brings us to the in-slice node $s4$, and $t2$ is added because if we leave it out, we can only reach in-slice node $s3$ by going through $s4$ first, even though we can reach it without passing here in the original.

When this restoration algorithm is applied, the resulting slices will adhere the *strong* correctness property[AAC13], which means that the following two statements hold for the resulting slice:

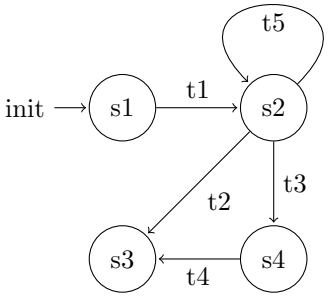
- If the original model shows certain behavior with respect to a variable in the slicing criterion, then the sliced model can show the same behavior
- If the sliced model shows certain behavior with respect to a variable in the slicing criterion, then the original program can show the same behavior

One downside of this method is that it can result in nodes not relevant to the slicing criterion being in the final slice. This gives us understandable traces, but in theory it can result in significantly larger slices: in our example we end up with almost the full original slice: only $t5$ is removed. To minimize the effects of adding these nodes to the slice on solver time, we remove all effects from events added to the slice during this step. We only add the guard node fore very event to the slicing criterion. From here, we slice again, reconnect again, and iterate until no new nodes are added in the reconnecting step. This process terminates, because eventually no new nodes will be added, or all nodes are in the slice. Either way, the slice stabilizes and our algorithm terminates.

Another way to minimize the effects of adding unnecessary nodes is to apply our second method

Figure 9: Graphical example of a specification and its slice

(a) Original specification



(b) Specification after slicing

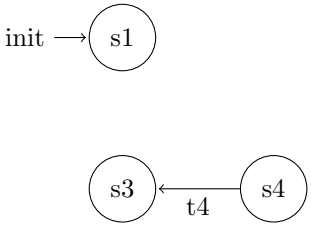
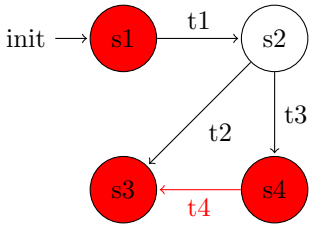


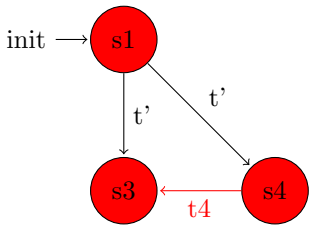
Figure 10: Specification from slice in figure 9 after reconnection step. All states and transitions that are part of the slice are displayed in red.



```
input  : A CFG G = (N, E) and a set of in-slice nodes S
output: A set missing_nodes containing all CFG nodes that need to be added to the slice to
        restore reachability relations
begin
    discovered ⟵ ∅;
    missing_nodes ⟵ ∅;
    forall n ∈ S do
        discovered ⟵ ∅;
        missing_nodes ⟵ missing_nodes ∪ DFS(G, n);
    end
    return missing_nodes;
end
function DFS(G,n):
    discovered ⟵ discovered ∪ {n};
    missing ⟵ ∅;
    foreach m ∈ σ(n) do
        if m ∉ S then
            missing ⟵ missing ∪ path_to_next_part_of_slice(CFG, m);
        end
    end
    return missing_nodes;
function path_to_next_part_of_slice(CFG,n):
    discovered ⟵ discovered ∪ {n};
    missing ⟵ ∅;
    next ⟵ ∅;
    if n ∈ slice then
        next ⟵ next ∪ {n};
    end
    else
        foreach m ∈ σ(n).m ∉ discovered do
            missing ⟵ missing ∪ path_to_next_part_of_slice(CFG, m);
        end
        if next ≠ ∅ then
            missing ⟵ missing ∪ {n};
        end
    end
    return missing;
```

Algorithm 3: Restoring reachability

Figure 11: Specification from slice in figure 9 after amorphous reconnection step. All states and transitions that are part of the slice are displayed in red.



of *amorphous* reconnection. As we learned in section 2, amorphous slicing results in a slice that is not necessarily a subgraph of the original: new edges may have been introduced. The idea is that when reachability between two in-slice node is not the same as it is in the original, new edges created between these in-slice nodes are added. Figure 11 shows us the result for applying this method to the slice we saw in figure 9. Algorithm 4 shows us the computations for amorphous graph reconnection.

Slices computed in this way adhere only the *weak* correctness[AAC13] property. Only one of the statements that make up strong correctness holds for slices reconnected amorphously:

- If the original model shows certain behavior with respect to a variable in the slicing criterion, then the sliced model can show the same behavior

The other property required for strong correctness does not hold. This means that behavior that can be witnessed in the slice may not exist in the model. As a result, model checking with amorphous slicing may produce counterexamples that do not exist in the original model.

Another obvious downside of this method is that error traces produced in amorphous slices do not translate directly to error traces in the original specification. If not just verification, but error tracing is the main goal, we recommend not using amorphous slicing. If, on the other hand, the user is not interested in a trace, we recommend applying amorphous slicing, as it can have a big impact on the size of the specification that is used for checking.

4.6 Reconstructing specifications from slices

Once we have obtained a slice, we need to transform it back into a Rebel specification so that it may be presented to the verification engine. Recall that a slice consists of a set of instances: pairs of variables and nodes. Let V be the set of all variables occurring in the slice, and N all nodes. That is, $V = \{v \mid (n, v) \in slice\}$ and $N = \{n \mid (n, v) \in slice\}$. We obtain the sliced specification from the original by traversing and altering it top-down. First, we check for every field, whether it there is a corresponding variable in V. If there is not, we remove this field from the specification.

Next, we check for every event, whether its corresponding guard node is in the slice. If so, we keep the event and traverse its definition. For every postcondition and synchronization statement we check if the corresponding node occurs. If not, we remove this part of the event definition.

Finally, we inspect the specification's life cycle. For every entry `s1 -> s2 : e1, ..., en`, we check if any nodes corresponding to the events are in the slice. We remove any event that isn't in the slice from the life cycle entry. If none remain at the end of this step, we remove the entire entry.

If we are applying amorphous slicing, one additional step remains: in addition to removing irrelevant parts from the specification, we may also have to *add* transitions to the specification. In this case, we add a single event `e'` to the set of allowed events. The event definition is empty: it contains no preconditions, postconditions and synchronization statements. For every two states having to be connected, we add an entry connecting them through `e'` to the life cycle.

This, finally, leaves us with a Rebel specification that can be sent to the verification engine.

5 Implementation

The algorithms presented in section 4 are implemented in Rascal[6]. This language was designed specifically for metaprogramming[KVDSV09a; KVDSV09b], and it's the language that Rebel was built in. The slicer depends on Rebel's own building module for parsing source code, importing relevant specifications and solving references. The graphs used in the algorithm for slicing and supporting algorithms were implemented using Rascal's built-in graph data type.

The output created by the slicer— a (set of) Rebel specification(s)— is placed in a subfolder called "slices", which is placed in the same folder as the original specification. From here, it can be treated just like any ordinary specification. From here, for instance, the verification engine can be called to check the TestModule that the slice was built for.

6 Evaluation

In this section we evaluate the slicing tool introduced in the previous sections. We introduce a set of benchmark specifications and compare the

[6]More information and download can be found at https://www.rascal-mpl.org.

```
input  : A CFG G = (N, E) and a set of in-slice nodes S
output: A set missing_edges containing edges between in-slice nodes that need to be added to the
        graph to restore reachability relations
begin
    discovered ⟵ ∅;
    missing_edges ⟵ ∅;
    forall n ∈ S do
        discovered ⟵ ∅;
        missing_edges ⟵ missing_edges ∪ DFS(G, n);
    end
    return missing_edges;
end
function DFS(CFG,n):
    discovered ⟵ discovered ∪ {n};
    missing ⟵ ∅;
    foreach m ∈ σ(n).m ∉ discovered do
        missing ⟵ missing ∪ {(n, v) | v ∈ find_next_part_of_slice(CFG, m)};
    end
    return missing;
function find_next_part_of_slice(CFG,n):
    discovered ⟵ discovered ∪ {n};
    next ⟵ ∅;
    if n ∈ slice then
        next ⟵ next ∪ {n};
    end
    else
        foreach m ∈ σ(n).m ∉ discovered do
            next ⟵ next ∪ find_next_part_of_slice(CFG, m);
        end
    end
    return next;
```

Algorithm 4: Restoring reachability amorphously

results of verification time compared to unsliced models. We discuss the results of benchmarking and enclose additional, unexpected findings.

6.1 Rebel specifications for benchmarking

To test the effectiveness of our slicing tool we should benchmark it against unsliced specifications. Unfortunately, the amount of TestModules (or specifications, for that matter) available at the time of writing is small. The ones that exist are of limited size. Rebel is still very much under development, and hence not a lot of work has been done in modeling actual business entities.

Thus, in order to test whether our tool allows verification of larger scale modules, we will have to find specifications elsewhere. Ideally, we would test our tool on existing benchmark sets, as this would show us how our tool holds up against the current state-of-the-art verification tools. Looking at the body of work on benchmarking BMC[BCC+03; CFF+01; Sht01; BLM01], however, we learn that these specifications tend to be proprietary. In order to measure the efficiency of our tool, we will have to create our own specifications. Since the process of constructing specifications for real world business entities with product owners at the bank is an iterative and time-consuming process beyond the scope of this work, we choose to create the specification for benchmarking from existing specifications. This allows us to construct test for different aspects of the slicer, and it helps us show theoretical best and worst cases. A major downside of this approach is that the best cases may not be applicable to specifications of actual product at all, since these may be structured very differently.

The specifications we have created for benchmarking our tool can be sorted into two different classes, each testing a different aspect. In one specification, a bug (or goal state) occurs increasingly deep. This allows us to measure the effects of model size on deep searches.

The second set of benchmark specifications increases a model's size without increasing the size of parts relevant to the goal state. This tests the effect of different full model to relevant sub-model ratios on model checking time. Both sets of specifications are treated in more detail below.

6.1.1 Benchmark I: increasingly deep bugs

To test the effect of search depth to model checking time, we construct a set of specifications in which the goal state is located at increasing depths. The search bound is increased accordingly. We create a familiy of models. The models we construct are simple: each contains n integer fields, $V_1, V_2, \ldots V_n., n \in \{1, 2, 4, 6\}$. The models contain only two states: an initial state `s1`, and a second, final state `s2`. The transition from the initial state to the second state is parameterized with an integer. It initializes all fields to the supplied integer value. From `s2` to itself, we define the looping events $\texttt{decrementV}_1$ to $\texttt{decrementV}_n$. Each of these events decreases the value of its corresponding field by 1. The code for this specification is shown in figure 12. A graphical representation of this specification is shown in figure 14.

Figure 12: Rebel specification of benchmark I

```
specification BMI {
    fields {
        id : Integer @key
        v1 : Integer
                ⋮
        vn : Integer
    }

    events {
        initialize[startAmount = m]
        decrementV1[]
                ⋮
        decrementVn[]
    }
    lifeCycle {
        initial s1 -> s2 : initialize
        s2 -> s2 : decrementV1, ...,
     decrementVn
    }
}
```

For slicing we create a TestModule describing a goal state in which the field v_1 reaches a negative value. This module is shown in figure 13. By increasing the initial value of v_1, we force the model checker to search deeper. We benchmark with maximum search depth starting from 10 and increasing up to 25. These values were chosen experimentally by increasing search depth until solving the unsliced specification started taking over 11 hours. We run one set of tests for every model obtained by creating n fields and corresponding events, for $n \in \{1, 2, 4, 8\}$:for every set the size of the sliced model doubles.

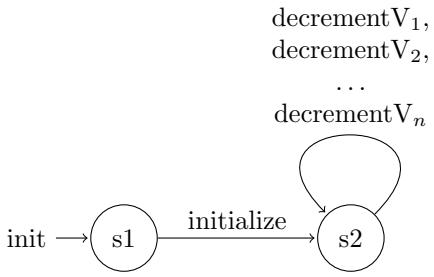
Figure 13: Slicing criterion for bencmark I, where the initial value for v_1 is set to n

```
state negV1 {
  s2 DeepSearch with V1 < 0;
}
check negV1 reachable in max n + 2 steps;
```

6.1.2 Benchmark II: incrementing model size

The second set of benchmark specifications maintains a constant depth for the goal state. Instead, it tests the effects of verifying models that are increasingly large compared to their sub-parts relevant to the slicing criterion.

Figure 14: Graphical representation of the specification for benchmarking with increasingly deep goal states



For this set of benchmarks, we extend the `Account` entity to contain contain additional copies of every field, event and state (apart from the initial state, of which there can be only one). If n is the number of copied structures in the model, figure 15 shows us the case where $n = 1$. For every iteration, we double the model's size. We check for $i = 0$, $i = 1$ and then up to $i = 63$. This bound was chosen because working with larger specifications turned out to be infeasible for Rebel's compiler. Note that any state, event, variable or field referenced in copy i will be suffixed with an i to prevent name clashes.

The TestModule constructed to slice and verify with looks for one of our extended `Accounts` with a negative `balance`. In the sub-model created for copy i, the `balance` field will have been renamed to `balance1`. Hence, the only part of the model relevant to the slicing criterion will be found in the original model. The slice will thus look exactly the same for every iteration.

6.2 Results

In this section we show the result of both sets of benchmarks introduced before. All experiments were run on an Intel Core i5 2.5Ghz machine running 64-bit Linux. Unless explicitly stated otherwise, all benchmarks obtained were from repeating a test 10 times and taking the average of all obtained result. This was done in order to account for the fact that Z3 employs a small amount of randomness in solving SMT-formulas.

6.2.1 Results I: increasing search depth

Table 1 shows us the results of benchmarking with increasingly deep goal states. We can see on one side the time Z3 required to solve the original specification. Under the column marked 'sliced C' we see the total of the time Z3 required to model check the slice resulting from slicing on the variables in C, the time our tool required to slice the original and then build the slice. The average time required to slice the specification was 0.1768 seconds. The column marked 'Factor' gives us the total time required by our tool divided by the time required by the original slicer.

The average time it took to build the slice was 0.5541 seconds. To gain more insight into how our tool performs compared to regular model checking, the results are plotted in figure 16.

6.2.2 Results II: increasing model size

We see the results for benchmarking with increased model size in table 2. This table is indexed by n, which signifies the model's size compared to the original specification. It shows us the time required for model checking the original, the time required to slice a specification of factor n, and the total time needed to slice, build and model check the slice. The average time it took to build the slice was 1.535 seconds, and the average time to solve was 0.013 seconds. The results of model checking with and without using our tool to slice first are plotted in figure 17.

6.3 Discussion

In this section we interpret and discuss the results of benchmarking our slicing tool. We see that, in a general, the results confirm our expectations that slicing facilitates model checking on larger models or with greater bounds. This is in line with findings from other work[CFF+01; Sht01; BLM01], saying that BMC can extend the size of models and search depth that can be checked within a feasible amount of time. We do, however, encounter some unexpected data points. These will be discussed below.

Looking at the benchmarks for deep searches, we see that, for the majority of search depths, the sliced models outperform the original. The exceptions to this rule are found at lower depths. This is because the cost of model slicing and building the new slice weighs heavier than what can be gained from checking a smaller model.

What is surprising from the results deep searches is the large amount of outlying data points. We expected the total solver time to increase steadily with search depth, but instead we see that sometimes deeper searches are actually faster. For instance, the original model was checked almost twice as fast at depth 13 as it was at depth 12. Even more interesting is the result found at depth 22, where the solver is about 25 times faster than it is at detph 23. All of the sliced models display similar jumps in model checking time.

[7] This test was run only twice due to time constraints.

[8] This and all consecutive depths for this specification were solved only thrice.

[9] Timed out after 41956.39 seconds. None of the deeper searcher reached a result within this time frame.

[10] This and all consecutive depths for this specifcation were solved only once.

Figure 15: Graphical representation of an `Account` extended with 1 copy

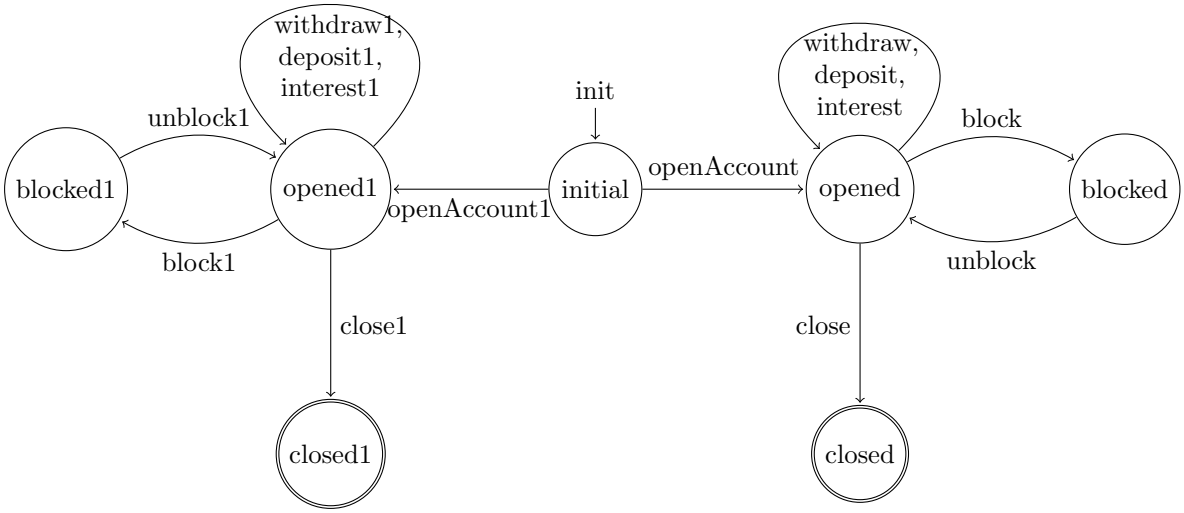


Figure 16: Plotted results for benchmarking deep searches

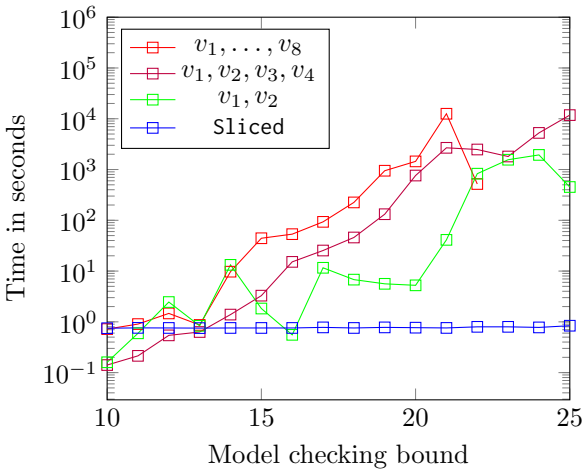
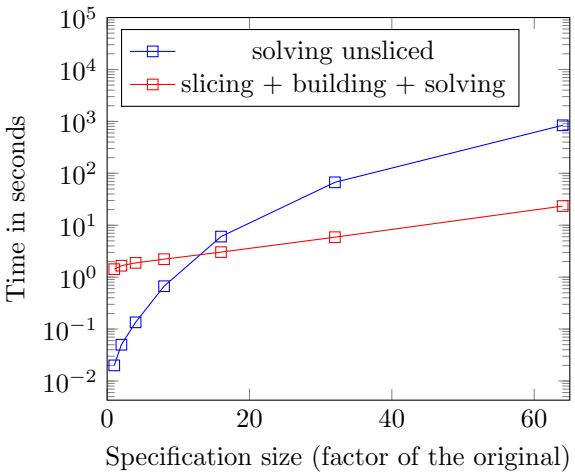


Figure 17: Plotted results for benchmarking `Accounts` of increasing size



One of the causes for these jumps in solver time may be concerned with formula order: the order in which entity events are declared in the SMT-LIB encoding sometimes varies per depth. Additionally, states in the original model are encoded by integer values in the formula encoding the specification. The integer values assigned to each state may differ. Although we have not been able to find literature on the effects of formula order on SAT- or SMT-solving time, we have found data showing other Z3 users experiencing different solver times due to changing formula order[11].

These minor differences in the SMT-encodings of our specifications do not always seem to explain the jumps in solver time, though: for the formulas created for the deep specification with respect to the variables v_1 and v_2, for instance, the order of events and the state encodings are the same, but solver time still makes some interesting jumps. We have no explanations for this.

Another result we did not fully expect is concerned with the factors between solving the original model and solving the sliced ones. We expected the differences in model checking times to scale with the differences in model size more predictably— i.e., for the slice with only the events `decrementV`$_1$ and `decrementV`$_2$, we had expected that the time required to check the model would about $\frac{1}{4}^d$—d being the search depth— of the time it would take to solve the original: at every decision point in the model, the slice has a quarter as many options as the original. This factor does not apply, and we assume that this is on account of one or more of Z3's optimizations.

[11] https://stackoverflow.com/questions/37237426/is-z3s-search-time-sensitive-to-formula-order

Table 1: Results for benchmarking deep searches. Each set of variables in the top row indicates which fields and corresponding transitions were in the benchmarked specification. The columns marked **SD** give the standard deviation of solver time.

	$v_1, \ldots, v_8$		v_1, v_2, v_3, v_4		v_1, v_2		**Sliced**		
Depth	**Time**	**SD**	**Time**	**SD**	**Time**	**SD**	**Solve**	**Total**	**SD**
10	0.721	0.009	0.141	0.003	0.16	0	0.021	0.749	0.003
11	0.904	0.003	0.214	0.008	0.595	0.007	0.033	0.761	0.005
12	1.466	0.029	0.542	0.004	2.455	0.021	0.03	0.758	0
13	0.869	0.003	0.631	0.211	0.835	0.007	0.025	0.753	0.005
14	9.727	0.465	1.39	0.043	13.26	0.057	0.03	0.758	0
15	44.321	0.203	3.28	0.023	1.83	0	0.03	0.758	0
16	53.344	0.413	15.179	0.065	0.555	0.007	0.03	0.758	0
17	93.212	0.494	25.398	0.745	11.605	0.007	0.05	0.778	0
18	224.951	4.704	45.86	0.628	6.785	0.0212	0.03	0.758	0
19	946.768	27.975	131.824	2.813	5.61	0.127	0.05	0.778	0
20	1446.41	33.77	761.46	21.4	5.24	0.014	0.04	0.768	0
21	12537.835[7]	358.213	2676.213	32.914	41.185	0.304	0.032	0.759	0.004
22	518.53	2.623	2479.203[8]	49.248	827.8	4.172	0.07	0.798	0
23	N/A[9]	N/A	1805.17	67.629	1559.62[10]	N/A	0.063	0.79	0.009
24	N/A	N/A	5260.59	71.953	1939.09	N/A	0.049	0.777	0.016
25	N/A	N/A	11802.32	440.686	452.47	N/A	0.111	0.839	0.02

Table 2: Results for benchmarking `Accounts` of increasing size. The column marked **SD** gives the standard deviation for the solver time. The standard deviation for solving the sliced specification was 0.005.

	Sliced			**Original**	
n	**Slice**	**Total**	**Factor**	**Time**	**SD**
1	0.127	1.426	71.3	0.02	0
2	0.167	1.681	33.62	0.05	0
4	0.335	1.884	13.956	0.135	0.005
8	0.791	2.219	3.3169	0.669	0.006
16	1.652	3.038	0.502	6.047	0.063
32	4.183	5.884	0.088	67.06	2.99
64	21.444	23.402	0.028	841.607	60. 71

The results from the benchmark set for `Accounts` of increasing size are less surprising, as there are no unexpected outliers to be found here. We see that initially, slicing is the more expensive procedure. But after we have increased the model size by a factor of 16, slicing becomes significantly faster. The total time required to wait for factor 64 is 23 seconds, whereas the original takes over 14 minutes to solve. We expect the difference in efficiency to increase with bigger factors.

The main reason we have not been able to test with bigger model sizes has to do with a surprising bottleneck: after a certain size, it became increasingly difficult for the IDE to cope. Models with 32 and 64 copies of the original `Account` specification took tens of minutes to be parsed and used. During this time, the IDE was unresponsive to other queries. Hence, working with models of this scale is not realistic.

The results answer, to some extent, the question of whether model slicing can be applied to optimize bounded model checking. Unfortunately, they do not tell us how our tool compares to one taking coarser dependences into account. We do not know if or at which point model slicing with this level of granularity outperforms the more traditional approach.

7 Conclusion

The work presented here was performed with the aim to answer the two questions stated below:

- Can we make model slicing more fine-grained?
- Can bounded model checking benefit from model slicing?

In our attempt to answer these questions, we created a slicing algorithm taking into account finer dependences than we have seen in other works on model checking. This algorithm is described in section 4. We implemented this algorithm into a model slicing tool for Rebel specifications, and compared the model checking time for sliced specifications with that of unsliced specifications. The results are demonstrated and discussed in section 6.

In the following two sections we answer our research questions separately based on the findings in the previous sections.

7.1 Can we make model slicing more fine-grained?

In section 4 we introduced a slicing algorithm which takes into account more fine-grained dependences than those we have seen in other model slicing algorithms so far. We did this by taking inter-variable dependences into account, and by creating multiple nodes for complex transitions. We saw that this algorithm requires additional types of dependences on top of the ones introduced in these aforementioned other works. We saw that these additional dependences require some extra calculations, but they are not inherently difficult to compute, and once the dependences are now and added as edges into the dependence graph, the same slicing algorithms can be applied as we have seen in other works. We have not benchmarked the resulting slices against sliced produced by algorithms relying on the coarser dependences that we see in other works.

To conclude: we can say that model slicing can easily be more fine-grained, but it should be noted that we do not know whether the calculations required for the additional dependences outweigh the cost of model checking slices produced by coarser model slicing algorithms.

7.2 Can bounded model checking benefit from model slicing?

In section 6 we benchmarked our slicing tool and compared the results against the model checking time required for unsliced specifications. We saw that the initial overhead of slicing was not worth the while for small slices and superficial counterexamples, but the results also showed that slicing was actually very beneficial on larger specifications and deep occurrences of goal states. The set of deep benchmarks showed that there is a very clear relation between model size and solver time as search depth increases, although we saw some outliers that we are not able to explain. We conclude that slicing may not always be beneficial, but in cases it can help to verify larger models and search with greater bounds within a given time-limit.

8 Future Work

During our research we encountered problems and questions that we could not solve within the scope of this work. In this section we list these questions as recommendations for future research, and supply potential starting points where we can.

8.1 Comparing our model slicing algorithm against coarser ones

In this worked we benchmarked the time required for model checking our fine-grained slicing algorithm against unsliced specifications. The application of our tools appears to be beneficial for the tests we performed, as it allowed us to check larger models and perform deeper searches. However, the algorithm has not been compared to the results of the coarser algorithms described in previous works on model slicing. It is yet to be seen whether the overhead of slicing more precisely is worth the effort. To this end we recommend either generalizing our tool to operate on different kinds of dependence graphs, or inversely to implement the algorithm we described into existing model slicing tools. Either way, the results should be benchmarked extensively.

8.2 Heuristics for combining criteria for multiple setup statements for the same entity

In section 4 we saw that, when multiple setup statements for a single entity occur in a TestModule, we have to take the the union of all induced slicing criteria. The reason we can not create separate slices for every setup statement, as we saw explained in more detail in the referenced section, is that it is not clear for any other entities referencing the entity for which multiple setup statements occur, which of the slices they should refer to. As we are forced to generate the largest, most general slice for all instances, we potentially end up with a larger search space than we would need if we could create separate slices We can think of theoretical examples where we could gain tremendous amounts of time by checking the smaller slices.

To combat this problem, we recommend researching whether it may be beneficial to analyze when the creation of separate slices does not induce the aforementioned problem. One way to go about this is to see whether the entity under consideration is referenced by other entities at all. Additional research might involve constructing heuristics for referring to the correct slice in the case when entities *are* referenced by other entities. This is likely to be non-trivial, as it requires deep insight into the criteria and the function of the referenced entities.

8.3 Additional benefits of model slicing

When exploring related work on the subject of (model) slicing, we saw that this technique is being used for a variety of applications in addition to model checking: it is used for debugging, comprehension, testing and promoting re-usability. We recommend exploring whether any of these applications could be advantageous for Rebel developers and product owners at ING Bank.

8.4 Additional optimizations for model checking

To further improve the efficiency with which bounded model checking can be performed, we

recommend implementing additional optimizations lateral to model slicing.

In section 2 we explored other approaches, and in addition to model slicing both the application of solver strategies and tuning the SMT-solver showed potential. We described related work on tuning SAT-solvers, which showed that assigning values to variables in order of their relative dependences can significantly speed up model checking time. Since our algorithm calculates these dependences anyway, we propose propagating the resulting ordering to the SMT-solver and changing Z3's source code to take it into account.

8.5 Better benchmarking: bigger, more realistic case study

At the time of this research, only a very limited set of specifications was available. For benchmarking the effects of increasing model size and search depth we have had to rely on fabricated specifications, but such artificial examples do not necessarily correspond to realistic specifications in underlying structure. To gain better insight into the applicability of our tool for optimizing bounded model checking in practice, we recommend extensively benchmarking it against a set of real world specifications. To this end, developers and product owners should come together to model their full systems.

8.6 Proving correctness

In this work we have not given a correctness proof for the model slicing algorithm we created. Although it is very similar to related model slicing algorithms which have been proven correct, and although the results of empirical evaluations are positive, a formal proof of correctness desirable. For this, we recommend looking at the correctness proofs present in other works on model slicing, which rely on the notion of simulation, and creating additional cases for the dependences that our algorithm takes into account.

References

[AAC13] Torben Amtoft, Kelly Androutsopoulos, and David Clark. Correctness of slicing finite state machines. *RN*, 13:22, 2013.

[ACH+13] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. State-based model slicing: A survey. *ACM Comput. Surv.*, 45(4):53:1–53:36, August 2013.

[All70] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. *Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers*, pages 146–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[BCM+90] J. Burch, E. Clarke, K. Mcmillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^20 States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[Bey14] Dirk Beyer. Status report on software verification - (competition summary sv-comp 2014. In *In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS*, pages 373–388, 2014.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BLM01] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 454–464, London, UK, UK, 2001. Springer-Verlag.

[Boe72] Barry W Boehm. Software and its impact: A quantitative assessment. *Datamation*, 19, 1972.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard – version 2.0. In *Proceedings of the 8^{th} International Workshop on Satisfiability Modulo Theories (SMT '10)*, July 2010. Edinburgh, Scotland.

[CCGR99] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *International conference on computer*

aided verification, pages 495–499. Springer, 1999.

[CDHW] Sylvain Conchon, David Déharbe, Matthias Heizmann, and Tjark Weber. Smt-comp 2016. http://smtcomp.sourceforge.net/2016/.

[CFF+01] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 436–453, London, UK, UK, 2001. Springer-Verlag.

[CGJ+01] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, UK, 2001. Springer-Verlag.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[dL01] Andrea de Lucia. Program slicing: Methods and applications. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 00:0144, 2001.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[dMP13] Leonardo Mendonça de Moura and Grant Olney Passmore. The strategy challenge in smt solving. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics*, volume 7788 of *Lecture Notes in Computer Science*, pages 15–44. Springer, 2013.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[JR94] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. *SIGSOFT Softw. Eng. Notes*, 19(5):2–10, December 1994.

[KLB12] Jochen Kamischke, Malte Lochau, and Hauke Baller. Conditioned model slicing of feature-annotated state machines. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD '12, pages 9–16, New York, NY, USA, 2012. ACM.

[KVDSV09a] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Easy meta-programming with rascal. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 222–289. Springer, 2009.

[KVDSV09b] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE, 2009.

[MPC+02] Madan Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *OSDI 02: Fifth Symposium on Operating Systems Design and Implementation*, page 75–88. USENIX, December 2002.

[Oja07] Vesa Ojala. *A slicer for UML state machines.* Helsinki University of Technology, 2007.

[Roz11] Kristin Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163 – 203, 2011.

[Sht00] Ofer Shtrichman. Tuning sat checkers for bounded model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 480–494, London, UK, UK, 2000. Springer-Verlag.

[Sht01] Ofer Shtrichman. Pruning techniques for the sat-based bounded model checking problem. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '01, pages 58–70, London, UK, UK, 2001. Springer-Verlag.

[Sil12] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012.

[Sin13] Rupinder Singh. Literature analysis on model based slicing. In *in International Journal of Computer Applications, ISSN 0975 – 8887, Volume 70– No.16*, 2013.

[SSVB16] Jouke Stoel, Tijs van der Storm, Jurgen Vinju, and Joost 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*, ITSLE 2016, pages 13–20, New York, NY, USA, 2016. ACM.

[WDQ02] Ji Wang, Wei Dong, and Zhi-Chang Qi. Slicing hierarchical automata for model checking uml statecharts. In *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '02, pages 435–446, London, UK, UK, 2002. Springer-Verlag.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.