Bachelor Informatica

University of Amsterdam

On the Use of SMT Solvers in Model-Based Testing

Florine de Geus

June 20, 2020

Supervisors: Machiel van der Bijl (Axini), Ana Oprescu (UvA), Alban Ponse (UvA)

Signed:

Acknowledgments

To say that these were unusual times to write a thesis would be an understatement, I suppose. I know that to some people, this might be 'just' a Bachelor thesis, but the whole process to me, from deciding on a subject to writing these acknowledgments was a unique experience in which I learned more than this text alone maybe represents. First of all, I would like to thank Machiel van der Bijl for reaching out to me and asking if I would like to do this thesis at Axini, along with great supervision during the process. I would like to thank Alban Ponse for his supervision during the project and for introducing me to the topic of formal methods in the first place. Furthermore, I would like to thanks Ana Oprescu for introducing me to Axini, helping Machiel and I with finding a subject that would both be suitable for a Bachelor thesis and interesting for Axini as a company, and lastly for stepping in for some last-minute supervision. I would like to thank Rob Belleman for coordinating the whole Bachelor project and making sure everything went as smoothly as possible in a completely online environment. Thank you to my friends for being great friends. Lastly, to my parents and sister: thank you for keeping up with me while I temporarily moved back with you during the lockdown and thank you to my dog (I wish she could read) for reminding me to go outside every once in a while.

Abstract

Testing of software systems is an essential aspect of the software development process. In recent years, focus has shifted from manual to automated testing and especially automated test generation. Model-based testing is a technique that automatically creates a test suite for a *Software Under Test (SUT)* by creating a model based on formal methods that describes the behavior. One approach to model-based testing is with a transition based system, where constraints over input and environment variables dictate whether a transition can take place or not. The generation of test cases is done by selecting paths within the system and verifying whether this path can be taken based on the constraints that are part of this path. If this is possible, the next step is to find suitable value assignments for all free variables (e.g. the variables that depend on external input) in the given constraints. There are a number of different approaches to solving such *Constraint Satisfaction Problems (CSPs)*. In this thesis, we explore the usability of SMT solvers in a model-based testing context. We evaluate our approach by analyzing the main considerations that need to be accounted for when implementing an SMT solver in a model-based testing context, as well as compare it to an already-existing approach that uses a Prolog-based constraint solver.

Contents

CHAPTER 1

Introduction

Testing of software systems is an essential aspect of the software development process. Insufficient test coverage could lead to unforeseen bugs or side effects that in some critical systems could have fatal consequences. However, developing tests is often seen as an expensive and tedious process and when the development of those tests is done manually, the risk of overlooking crucial parts is higher. This can, in turn, lead to high maintenance costs and in some cases, high societal impact.

In recent years, focus has shifted from manual to automated testing [22] and especially automated test generation. One method of automatically generating software tests is through model-based testing. Here, instead of writing software based on a specification given by the domain expert, a formal model that describes the behavior of the *Software Under Test (SUT)* is created. This model is then used to generate a set of suitable tests. There are several different approaches to employing model-based testing [32]. One of these approaches is by using a transition-based system, where transitions between different states describe different actions the system can take, based on a given input. These transitions can contain constraints that have to be met to be able to traverse to the next state. These constraints can, for example, be a numerical value that is required to lie within a certain numerical or a string that has to match a given regular expression.

The generation of test cases is done by selecting paths within the system and verifying whether this path can be taken based on the constraints that are part of this path. If this is possible, the next step is to find suitable value assignments for all free variables (e.g. the variables that depend on external input) in the given constraints. There are different approaches to solving such *Constraint Satisfaction Problems (CSPs)*. A classical method is by using backtracking and constraint propagation [19]. These techniques are typically used in logic programming languages like Prolog [15]. Another approach that has increased in popularity over recent years is by using *Satisfiability Modulo Theory (SMT)* solvers [20]. These solvers are extensions of the classic boolean satisfiability solvers or SAT solvers for short, and offer support to a wide variety of theories and data types, including integers, floating-point numbers, arrays, and strings.

In this thesis, we explore the usability of SMT solvers in a model-based testing context. We achieve this by exploring the ways an SMT solver can be used to generate test data, and implementing an existing SMT solver into a model-based testing context. We evaluate our approach by analyzing the main considerations that need to be accounted for when implementing an SMT solver in a model-based testing context, as well as compare it to an already-existing approach that uses a Prolog-based constraint solver. We primarily base our comparison on performance, as this is an important aspect to consider when using model-based testing in real-world applications. The central question we aim to answer is as follows:

How can we use SMT solvers in a model-based testing context?

We divide this research question into the following sub-questions:

1. How is constraint solving used in model-based testing?
2. What is the state-of-the-art in SMT solvers?
3. How can we use SMT solvers to generate test case data?
4. How can we implement an SMT solver in an existing model-based testing tool?
5. How does an SMT solver compare to other types of constraint solvers?

1.1 Justification

This bachelor's thesis was hosted at Axini. Axini is a company based in Amsterdam specialized in model-based testing for real-world applications. Axini provides the *Axini Modeling Suite*, a model-based testing environment based on formally defined transition methods. For this thesis, we were able to use their resources, including the Prolog-based constraint solver that is currently used to generate test case data to compare to the implementation presented in this thesis.

1.2 Ethical considerations

As mentioned in the introduction, testing is an essential aspect of the software engineering cycle. Having a set of tests that covers all essential parts of a codebase is of great importance in critical systems, but is difficult to achieve manually. Furthermore, improperly tested systems can have real-world consequences when a bug arises, for example in railway systems [1]. Applying formally proven methods such as model-based testing takes an important step toward fault-free systems. For industrial model-based testing tools, it is crucial that these tools achieve high testing coverage in a reasonable amount of time. While constraint solving plays only a small part in this landscape, having an efficient solver implementation can contribute to the efficacy and real-world employability of such model-based testing systems.

1.3 Outline

In Chapter 2, we will give a theoretical background of model-based testing using transition systems and the role constraint solving plays in generating test cases. Furthermore, we will give an overview of both Prolog-based constraint solvers as well as SMT solvers. In Chapter 3, we will discuss the SMT solver we will use in our implementation and how we plan to generate test data using this solver. Chapter 4 will then address how this SMT solver will be implemented in practice, and how we will compare our implementation to an already-existing solver based on Prolog. We will furthermore discuss how our implementation can be integrated into a model-based testing context in this chapter. In Chapter 5, we will provide the results to our comparison and analysis, which we will discuss in Chapter 6. In this chapter, we will also reflect on our research questions and give recommendations for future research.

[1] `https://www.prorail.nl/nieuws/oorzaak-computerstoring-treinverkeersleiding-gevonden`

CHAPTER 2

Model-based testing

In this chapter, we will discuss model-based testing in general, and give a theoretical background to a model-based testing technique that uses transition systems. Furthermore, we will discuss constraint solving in general and how it is used in model-based testing.

The concept of model-based testing, or MBT for short, was first introduced in the seventies [5, 16] and was based on finite-state machines. Since then, different approaches have been proposed that can be classified using the taxonomy proposed in [32]. A widely used approach to model-based testing uses transition systems, which have been introduced as Labeled Transition Systems by Jan Tretmans [31] and have been extended to Symbolic Transition Systems [13].

2.1 Labeled Transition Systems

A Labeled Transition System describes a system consisting of states with transitions. The transitions contain labels that describe the actions the system can take. Formally, a Labeled Transition System is defined as follows:

Definition 2.1.1. A *Labeled Transition System (LTS)* is a 4-tuple $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$, where

- S is a countable, non-empty set of states;
- $s_0 \in S$ is the initial state;
- Σ is a countable set of observable action labels;
- $\rightarrow$ is the transition relation from one state to another.

Generally, applications communicate through inputs and outputs. By dividing the labels in L into *input labels* Σ_{in} and *output labels* Σ_{out}, where $\Sigma_{\text{in}} \cap \Sigma_{\text{out}} = \emptyset$. When such systems are visualized, input labels are usually denoted with '?' and output labels with '!'. The action label τ denotes an unobservable (internal) action. For brevity, the union of Σ and $\{\tau\}$ is often denoted as Σ_τ. Furthermore, the notion of *quiescence* is introduced. A *quiescent state* is a state that does not perform any output actions [31]. A quiescent state s in an LTS is denoted by $\delta(s)$. Additionally, we can define a quiet action $\delta \notin \Sigma_\tau$. The union of Σ and $\{\delta\}$ is denoted as Σ_δ.

To further clarify the concept of Labeled Transition Systems, we introduce the following example. Consider a food vending machine. A customer can purchase food from this machine by inserting coins into a slot and pressing a button next to the food item of their choice. When the correct amount of money has been inserted and the button has been pressed, a mechanism unlocks the door the food is kept behind and the customer can take their order.

Suppose the food in our vending machine costs €2.00 and only €0.50, €1.00 and €2.00 coins are accepted. The resulting LTS is shown in Figure 2.1. Because the order in which the coins are inserted matters, we have to take into account every permutation of the sequences of coins that sum to €2.00, which are six permutations in total. As can be seen in Figure 2.1, we need 15 states to represent this system. If we were to also allow, for example, €0.20 coins, the number

of permutations then becomes 34, which leads to a total of 180 states. This phenomenon, where even a small increase of a system exponentially increases its number of states is called *state space explosion* [8].

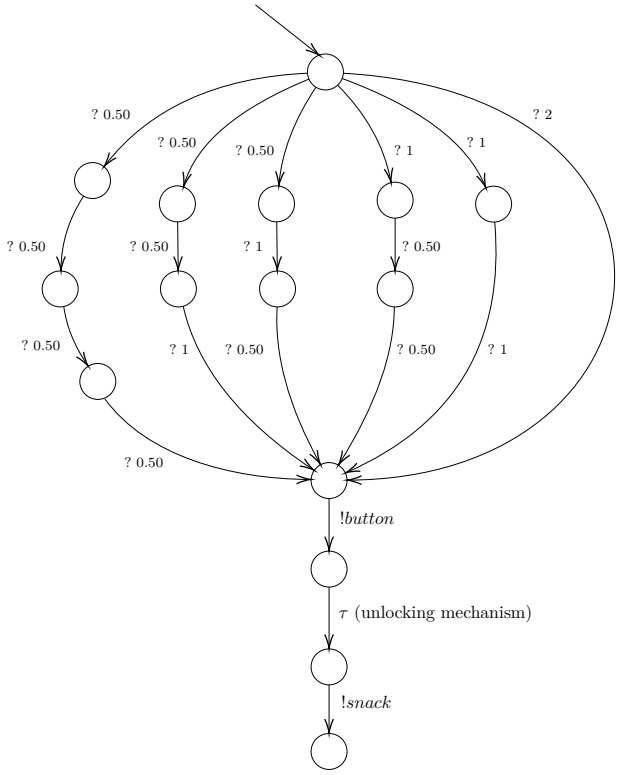

Figure 2.1: Example of a Labeled Transition System, illustrating the process of purchasing food from a food vending machine.

2.2 Symbolic Transition Systems

A solution to the state space explosion problem with Labeled Transition Systems, is to introduce the notion of data and variables. This results in Symbolic Transition Systems.

Definition 2.2.1. A *Symbolic Transition System (STS)* is a 7-tuple $\mathcal{S} = \langle L, l_0, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow \rangle$, where

- L is a countable set of locations;
- $l_0 \in L$ is the initial location;
- $\mathcal{V}$ is a countable set of location variables. These variables are accessible through the whole system;
- ι is an initialization of the location variables;
- $\mathcal{I}$ is a set of interaction parameters, where $\mathcal{V} \cup \mathcal{I} = \emptyset$. These parameters are only accessible by the switch in which they appear. To distinguish these parameters from location variables, we denote them with an underscore;
- Λ is a countable set of observable gates. The unobservable gate is τ. For brevity, the union of Λ and τ is often denoted as Λ_τ;
- $\rightarrow$ is the switch relation. This relation contains a *constraint* φ that contains variables from $\mathcal{V}$ and $\mathcal{I}$. This constraint has to be evaluated as *true* to be able to transition to the next location. It furthermore contains an *update mapping* ρ that updates the values in $\mathcal{V}$.

An STS based on the example discussed in Section 2.1 is shown in Figure 2.2. To ensure that only coins of €0.50, €1.00, and €2.00 are accepted, the constraint for *_coin* from the second location back to the first one has been set to $\_coin \geq 0.50$. To expand this to also accept €0.20 coins, we can simply modify this constraint, without having to add extra locations.

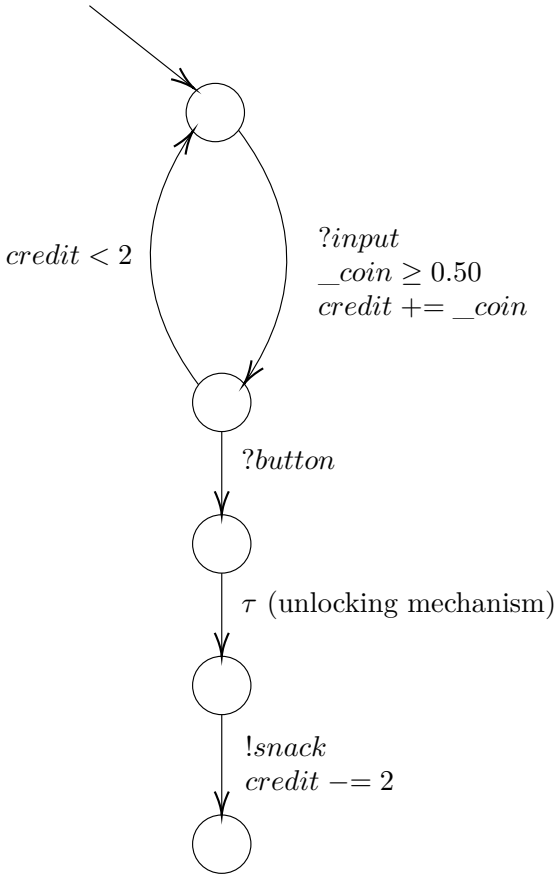

Figure 2.2: Example of a Symbolic Transition System, illustrating the process of purchasing food from a vending machine.

2.3 Input-output conformance

To verify whether a certain implementation conforms to the specified model, the implementation relation **ioco** [31] is applied. Before we give the definition of the **ioco** relation, we need to introduce some additional definitions relevant to **ioco**. These definitions are taken from [31]:

Definition 2.3.1. Let $p = \langle S, s_0, \Sigma, \rightarrow \rangle$ be a labeled transition system, then Σ_δ^* is the set of all finite sequences over Σ_δ and $\sigma \in \Sigma_\delta^*$ is called a *trace* that may contain quiescence. The set of traces of p that may contain quiescence is denoted as $Straces(p)$. Lastly, the set of states that are reachable from a state s through trace σ is denoted by s **after** σ.

Definition 2.3.2. Let s be a state in a labeled transition, then $out(s$ **after** $\sigma)$ gives all possible outputs occurring after having performed the trace $\sigma \in \Sigma_\delta^*$.

Using Definitions 2.3.1 and 2.3.2, we can define the **ioco** relation. Again, this definition is taken from [31]:

Definition 2.3.3. Given a set of input labels Σ_{in} and a set of output labels Σ_{out}, the relation **ioco** between a specification s and an implementation i is defined as follows:

$$i \textbf{ ioco } s \Leftrightarrow_{\text{def}} \forall \sigma \in Straces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$$

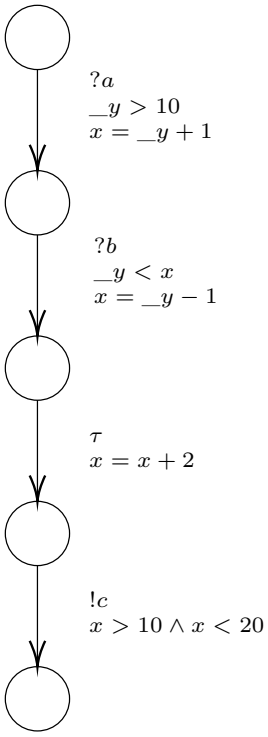

Figure 2.3: Example of a trace for some symbolic transition system.

To support symbolic transition systems, the **sioco** [12] relation is introduced. This relation does not change **ioco**, but rather lifts its definition to allow the notion of data and variables.

2.4 Test case generation

A challenge in generating test cases is ensuring that the SUT has been tested thoroughly within some finite amount of time. Certain coverage criteria can be used to verify whether a SUT has been sufficiently tested. A possible approach to this problem is by using a *Trace Coverage Strategy* [14]. Here, a finite set of traces from the model is constructed. For each trace, a trace-constraint is constructed. For each transition in the trace, this trace-constraint takes into account both its update mapping and its constraint.

By executing the algorithm given by [14], a trace-constraint is constructed by traversing the transitions in the trace and building a tree-like structure that represents the trace-constraint. The constraint structure corresponding to the trace given in Figure 2.3 is shown in Figure 2.4. This tree corresponds to the trace-constraint

$$((((\_y_1 - 1) + 2 > 10) \ \wedge ((\_y_1 - 1) + 2 < 20)) \ \wedge (\_y_1 < \_y_0 + 1)) \ \wedge \_y_0 > 10.$$

Here, $\_y_0$ refers to the first interaction parameter $\_y$ and $\_y_1$ to the second parameter. Keep in mind that these parameters are local to their transition, so they are renamed to distinguish between different parameters of the same name.

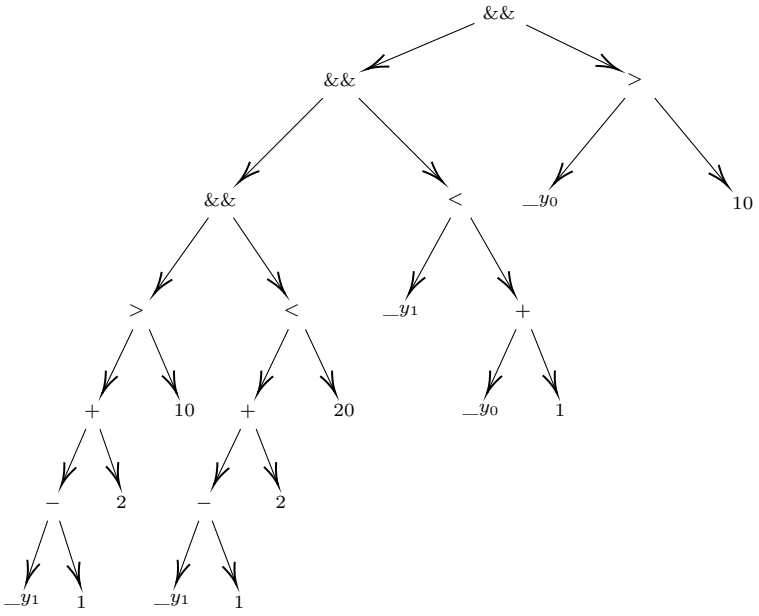

Figure 2.4: Tree representation of the trace-constraint corresponding to the trace given in Figure 2.3

By treating the trace-constraint as a *Constraint Satisfaction Problem (CSP)* [4], we can determine if a trace can take place at all by determining whether a solution exists for the given constraint. A CSP is defined as follows:

Definition 2.4.1. An instance of a CSP is a triple $(V, A, \mathcal{C})$, where

- V is a set of variables;
- A is a domain of values;
- $\mathcal{C}$ is a set of constraints $\{C_1, \ldots, C_n\}$. Each constraint $C_i \in \mathcal{C}$ is a pair $\langle s_i, \rho_i \rangle$, where
  - s_i is a tuple of variables of length m_i, called the *constraint scope*;
  - ρ_i is an m_i-ary relation of A, called the *constraint relation*.

The question that we want to answer then becomes:
Does there exist a solution, i.e. a function f, from V to A, such that for each constraint $\langle s_i, \rho_i \rangle \in \mathcal{C}$, with $s_i = (x_{i_1}, \ldots, x_{i_m})$, the tuple $(f(x_{i_1}), \ldots, f(x_{i_m}))$ belongs to ρ_i?

If no solution exists for the given trace-constraint, the trace can be omitted from the test suite. If a trace can take place, we can use a constraint solver to find a set of values for each of the interaction parameters for which the constraint is satisfiable. From each set, the minimum, maximum, and a random value is chosen [24]. These values are then used to generate test cases. Preferably, a test case consists of the boundary values of a variable (either minimum or maximum), as using boundary values of a test set has proven to generally be most effective [26]. However, it is not always possible to assign a boundary value to all variables in a constraint and have it still be satisfiable. In that case, a random value is assigned to one or more variables to make the constraint satisfiable again.

2.5 Constraint solving

A constraint solver is needed to evaluate the trace-constraints and generate suitable test data. While there are many approaches to solving given sets of constraints, each with their own types of applications, model-based testing tools typically use *Constraint Logic Programming (CLP)* or SMT solving to generate test cases.

2.5.1 Constraint Logic Programming

Constraint solvers using CLP initially got their origin in artificial intelligence, but quickly found popularity in other fields, including test model-based testing [17]. Typically, constraints solvers using CLP solve constraints over finite domains. The first methods of solving a set of constraints meant simply generating every possible set of solutions and checking if it satisfied all constraints [19]. Over the years, these methods have become more and more sophisticated, which has greatly improved the performance of these solvers. Modern constraint solvers use a technique called *constraint propagation*. Here, the domains of constraints and the constraints itself are repeatedly reduced until ultimately a set of possible solutions is found that satisfy the original set of constraints. To achieve this, a primitive constraint $X \in r$, where X is a variable and r a (non-empty) finite set of natural numbers is used as a starting point [9]. This automatically implies that the values of the given solution have to lie within r, and that the constraints have to be over natural numbers.

2.5.2 SMT solving

Another approach to solving CSPs is by using satisfiability modulo theory (SMT) solvers. These solvers are an extension of boolean satisfiability (SAT) solvers and support a number of theories, including integers, reals, arrays, and strings [20]. While SAT (and by extension SMT) problems are typically NP-complete, state-of-the-art algorithms have been developed that make it possible to handle such problems in an industrial context [23]. Most modern SAT solvers employ some variation of the *Davis-Putnam-Logemann-Loveland (DPLL)* algorithm, and early SMT solvers employed a technique called 'bit-blasting', where every constraint would be translated to a boolean expression and passed along to a classical SAT solver. More modern solvers, however, use a more general variant of the DPLL algorithm, called DPLL(T) [23]. Furthermore, most state-of-the-art SMT solvers combine different methods to increase performance [2, 11, 21].

CHAPTER 3

Designing a constraint solver for model-based testing

In this chapter, we will discuss the features required for a constraint solver to be used for model-based testing. Using these requirements, we will choose an SMT solver and explain how we will use the capabilities of our chosen solver to meet these requirements.

3.1 Main requirements

For the generation of data for test cases as described in Section 2.4, the constraint solver does not need to be able to return all solutions to a set of constraints. Rather, it should at least be able to find the boundary values as well as random values that satisfy the given constraints. In a finite domain solver such as GNU Prolog, it is trivial to find the boundary values, because the constraint solver returns the whole range of values that satisfy the given constraint. We then only need to select the smallest and largest value in that range. Finding boundary values using an SMT solver is generally more difficult, as they were initially meant to only determine whether a constraint is solvable in the first place. However, with the growing number of applications requiring the free variables in a constraint to be assigned non-arbitrary values, research into achieving this has increased in the last few years. An extension to SMT, Optimization Modulo Theories (OMT), deals with optimizing solutions to constraints [29]. The constraint solver we will use therefore has to support optimization.

Since model-based applications for industrial systems often support different data types and structures, the solver should be capable of supporting and combining different theories. Another point of importance, which we only touched upon in the introduction of this thesis, is the performance of the solver. Since model-based testing in practice is generally used in large-scale and critical applications [22], it is of importance that tests are generated as fast as possible, and using a fast constraint solver can contribute to this. Ideally, it should be able to match or outperform existing implementations, but at best it should be able to solve a constraint and get the desired solutions within a tenth of a second.

3.2 Selecting an SMT solver

While there are several SMT solvers currently being developed and maintained, only two solvers are available that support optimization: Z3 [20, 3] and OptiMathSAT [30], which is an extension to the general SMT solver MathSAT 5 [7]. These solvers share a number of similarities. They both offer bindings several programming languages as well as a stand-alone command line interface. Both interfaces support SMT-LIB as an input format. SMT-LIB [1] is a standardized language for SMT solvers that represents constraints and expressions as Common-Lisp S-expressions.

The main difference between Z3 and OptiMathSAT (and MathSAT in general) is its extensiveness. Z3 offers support for far more theories and data types, including strings and user-defined data structures. This makes Z3 more versatile for different applications and more suitable for model-based testing systems with custom data types. Considering this, we will use Z3 in our implementation. For the remainder of this chapter, we will explain how we will use Z3 to meet the requirements as stated in the section above. Throughout the chapter, we will give examples using the SMT-LIB syntax. An example of a constraint expressed using SMT-LIB is given in Figure 3.1. Here `(check-sat)` command verifies whether the given constraint can be satisfied, after which `(get-value (X))` returns a value that satisfies the constraint. Keep in mind that this does not yet use the optimization features; we will touch on that in the next section.

```
1  (declare-const X Int)
2
3  (assert (and (> X 0) (< X 12)))
4
5  (check-sat)
6  (get-value (X))
```

```
Output:
sat
((X 11))
```

Figure 3.1: SMT-LIB representation of the constraint $0 < X < 12$.

3.3 Adapting Z3 to model-based testing

There are a few requirements that need to be met for Z3 to be usable as a constraint solver within AMS. In this section, we discuss what those requirements are and how we plan to meet them.

3.3.1 Support for different valuation methods

As mentioned previously, to generate test cases we need the boundary (minimum and maximum) values of the variables in a constraint for which it can be satisfied, as well as random values. Finding the boundary values and a random value requires different approaches, so we will discuss them separately.

Finding boundary values

As mentioned in the previous section, Z3 offers support for such optimizations [3]. Using either the `minimize` or `maximize` command, we can find the optimal solution for the given variables. An example using `minimize` is given in Figure 3.2.

Z3 offers different priority options for optimizing constraints containing multiple variables [3]. By default, it uses a lexicographical priority. This means that it will solve the objectives in the order of declaration. For example, in Figure 3.2, X is minimized first, which yields the result $X = 0$. After that, Y is minimized. To satisfy the given constraint, Y is evaluated to $Y = 10$. If we were to minimize Y first, the results would be inverted. To optimize every variable in a constraint independent from each other, we can use the `box` option. This would result in X and Y both being assigned 0. Lastly, optimization objectives can be prioritized using Pareto fronts, where every variable is optimized as best as possible. Note that this will initially lead to the same solution as with the lexicographical priority. However, if we check the assertions again, X and Y are assigned 1 and 9, respectively, whereas the lexicographical priority will always result in the same objectives. In theory, we could use the Pareto prioritization to obtain all possible

```
(declare-const X Int)
(declare-const Y Int)

(assert (= (+ X Y) 10))
(assert (>= X 0))
(assert (>= Y 0))

(minimize X)
(minimize Y)

(check-sat)
(get-objectives)
```

```
Output:
sat
(objectives
  (X 0)
  (Y 10)
)
```

Figure 3.2: Finding the minimum X and Y that satisfy the constraint $X + Y \geq 10 \wedge X \geq 0 \wedge Y \geq 0$ using Z3's optimization feature.

combinations of objectives that satisfy a given constraint. However, this would require a call to the solver for every possible combination. This is not viable for constraints with a large number of free variables or variables with a large solution space. Furthermore, this is unnecessary for the method of test case generation discussed in this thesis.

Considering the different prioritization methods, the default lexicographical priority seems most suitable for the method of test case generation as discussed, because it ensures that at least one variable gets optimized. Furthermore, often the majority of variables in the constraints in an STS are independent of each other. This means that lexicographic optimization is sufficient most of the time.

Finding random values

In contrast to finding boundary values, Z3 does not offer native support for finding randomized solutions to given constraints. There are two possible methods we can implement to still be able to do so. The first one is to use the Pareto front as discussed above to generate a set of solutions from which we randomly pick one set. This would not be ideal, since it requires multiple calls to solver by default. Therefore, we propose a second option where we utilize Z3's optimization feature. By minimizing and maximizing a variable at the same time, we can find the complete range of each variable in the constraint. For each variable, we then pick a random value between that range and add a constraint assigning that value to the variable and check the satisfiability of these constraints. If the solver returns `sat`, it means we've found valid random values for each variable. If the solver returns `unsat`, we repeat the above steps until we find a set of variable assignments that does satisfy the given constraints. To prevent repeating this loop with sets of assignments from which we already know are unsatisfiable, we keep track of these sets so we can exclude them when finding new random assignments. See Algorithm 1 for the full algorithm. By using this algorithm we will make two calls to the solver in the best case, and $1 + n$ calls in the worst case, where n is the number of possible combinations of random assignments. For unbounded constraints, this means that this algorithm can run infinitely long. However, as we mentioned before, most variables in the constraints of an STS are unrelated which makes the chance of this occurring relatively slim.

Algorithm 1 Finding random solutions for a set of variables V in a set of constraints C.

```
1: function FindRandomValues(V, C)
2:     for var ∈ V do
3:         Minimize(var)
4:         Maximize(var)
5:     end for
6:     FindRanges(V, C)                          ▷ Find the range for every variable.
7:     if unsat then
8:         return
9:     end if
10:    repeat
11:        for var ∈ V do
12:            a := Rand(min, max)                ▷ Find a random value in the range of v.
13:            C.Add(v = a)   ▷ Add a constraint that equals v to a to the existing constraints.
14:        end for
15:        if A ∈ E then      ▷ Continue if the set of assignments has been checked already.
16:            continue
17:        end if
18:        CheckSat(V, C)          ▷ Check if C is satisfiable with the random assignments.
19:        if unsat then
20:            E.Add(A)                ▷ Exclude this set of assignments in next iterations.
21:        end if
22:    until sat
23: end function
```

3.3.2 Support for different data types

Z3 offers a wide range of theories and data types. Modern applications rely on in- and output other than boolean and numerical values, such as strings [25], and more complex data structures [33]. Thus, besides boolean constraints and numerical constraints (e.g. constraints over integers and real numbers), our implementation should be able to at least support constraints over strings and data structures such as arrays, structs, and hashes.

Boolean and numerical constraints

Since SMT solvers are an extension of SAT solvers, a classical SAT solver is used to solve constraints over boolean variables. As for numerical variables, Z3 supports constraints over both integers and real numbers, along with a number of relational, logical and arithmetic operators [34]. In Z3, expressions are represented as nodes in an abstract syntax tree. Because trace-constraints are constructed as a tree structure, these constraints can be constructed in Z3 by traversing this tree.

An important thing to take into account are constraints using non-linear arithmetic. Although steps are being made into being able to solve these types of constraints over integers as well as real numbers [6], Z3 does not yet support non-linear integer arithmetic and only supports non-linear polynomial real arithmetic [18]. A method of handling constraints containing non-linear integer arithmetic could be to add a pre-processing step that rewrites these constraints to constraints over real numbers. Non-linear transcendental arithmetic can furthermore be approximated by taking their k-th Taylor polynomial instead. However, as this is beyond the scope of this thesis, for now, we will restrict our implementation to linear arithmetic only.

Constraints over strings

Z3 offers a specialized built-in solver for string constraints, *Z3-str* [35]. Contrary to single-purpose string solvers, Z3-str allows string constraints to be combined with other data types, which makes it highly suitable for model-based testing purposes. Besides checking string equality, Z3-str also supports more complicated string operations, such as sub-strings, regular expressions, and string

```
(declare-const S String)
(declare-const X Real)

(assert (= S "Hello, world!"))
(assert (and (>= X 3.14) (<= X 9.81)))

(maximize X)
(check-sat)
(get-value (S X)
```

```
Output:
sat
((S "Hello, world!")
 (X (/ 981.0 100.0)))
```

Figure 3.3: Example of combining different data types in Z3.

concatenation. For this thesis, we will limit ourselves to solving string constraints that check equality of a given string literal, as these comparisons are most common in model-based testing.

Constraints over data structures

While the implementation of constraints over (multi-dimensional) data types such as arrays, structs, and hashes are beyond the scope of this thesis, it is worthwhile to mention that Z3 does offer support for such data structures. Whereas arrays are natively supported by Z3, a *datatype* feature is offered to let users define their own (recursive) data types [34]. This feature can be used to create an abstraction layer for other data types, such as the ones mentioned here.

CHAPTER 4

Experimental design and setup

In this chapter, we will describe how we implemented the design proposed in the previous chapter. First, we will discuss the architecture of the model-based testing platform offered at Axini. Then, we will explain how we will implement Z3 to be able to integrate with the existing platform. Lastly, we will describe how we will compare our implemented constraint solver to the constraint solver that is currently deployed at Axini.

4.1 The Axini Modeling Suite

The Axini Modeling Suite offers a Domain Specific Language (DSL) called the *Axini Modeling Language (AML)*. It is used to describe the desired behavior of a system that handles observable input and output. AML supports different data types for both interaction parameters and location variables. The set of data types AML currently supports consists of integers, decimal numbers, strings (including regular expressions), dates, and times. Furthermore, it is possible to store these data types in (nested) lists, hashes, and structs. Lastly, a special *void* data type is supported to denote the absence of a value.

Based on the models specified using AML, test cases are generated for the *Software Under Test (SUT)*. A schematic overview of the structure in the AMS with regard to constraint solving is shown in Figure 4.1. When a model has been specified in the Axini Modeling Language, a symbolic transition system is generated from this model using the *STS-engine*. The constraints in this system are represented by Axini's constraint language *Dumont*. Dumont internally represents the defined constraints as a parse-trees, which are solved using *treeSolver*, which creates a GNU Prolog query from the parse trees and sends it to the constraint solver.

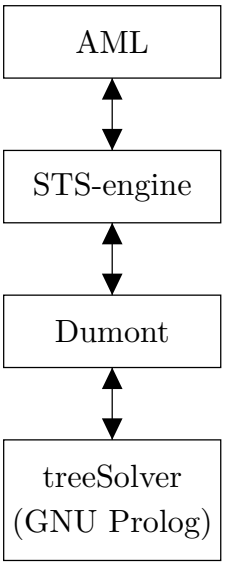

Figure 4.1: Structure of the Axini Modeling Suite.

4.1.1 GNU Prolog

GNU Prolog is a Prolog compiler that includes a constraint solver over finite domains [10]. Solutions to the variables in a set of constraints can be represented in two ways [9]:

- *MinMax*, where only the minimum and the maximum value of r are stored. This representation is used for intervals. An example constraint for which this representation would be used is $6 < X < 12$, where the resulting range would be $X \in \{7 \dots 11\}$.
- *Sparse*, where each value of the range is stored in a bit-vector. This representation is used when a 'hole' appears in the range, which for example is the case for the constraint $X < 6 \wedge X > 12$. The resulting range for this constraint would be $X \in \{0 \dots 5, 13 \dots \texttt{vector\_max}\}$, where `vector_max` is the size of the bit-vector.

Because GNU Prolog is a finite domain solver, it is only to solve constraints for values within the range $[0, \ldots, 268435455]$.

There are a few drawbacks to the use of GNU Prolog within AMS. The main drawback is its lack of support for values other than natural numbers. As mentioned, AMS supports different data types as well as some data structures. Currently, this is solved by encoding every data type into a positive integer. For example, a decimal number can be encoded by multiplying them by 10^d, where d is a set precision. While this makes it possible to solve constraints containing different data types, it also introduces restrictions. Decimal numbers cannot be multiplied or divided, because they cannot be decoded back to the correct value when those operations have been performed. Another example is with time values. These are encoded into seconds. Because of this, it is only possible to solve constraints for ranges within eleven days since ranges beyond that will cause overflow, as they exceed the domain of GNU Prolog.

4.2 Implementation of Z3

The Axini Modeling Suite is implemented in the programming language Ruby, a dynamically typed interpreted programming language [28]. Z3 itself does not offer Ruby bindings to the solver. A third-party Ruby interface to Z3 exists[1], but it is still in early stages of development. Furthermore, it provides features of Z3 we do not immediately need for our implementation, while at the same time it lacks some features we do need. Therefore, we decided to use Z3's native C API and the Ruby-FFI library [27] to implement our own Ruby interface to Z3, so it can be tailored to be used in model-based testing, and specifically in the Axini Modeling Suite. Our implementation, dubbed 'SMTreeSolver', creates a layer of abstraction around Z3's native C bindings and allows constraints to be built as abstract syntax trees, that can then be solved with a single method call. This allows the solver to be implemented relatively easily into the AMS environment, as the constraints are already represented as abstract syntax trees with Dumont. An example of a constraint implemented in SMTreeSolver is shown in Figure 4.2, which implements the constraint given in Figure 3.2.

4.3 Performance of SMTreeSolver

For our analysis of the SMTreeSolver, we will use Axini's treeSolver as a reference. We will compare both solvers on performance. For our implementation to be applicable in a real-world model-based testing tool such as AMS, the performance of our implementation should at least match that of treeSolver. To test this, we compiled a set of constraints and execute it on both the treeSolver as well as the SMTreeSolver implementation. Table 4.1 shows the types of constraints we will use in our comparison. The constant values will be randomly generated and we will measure the time it takes to solve each constraint, using different valuation methods. We will repeat this multiple times for each type of constraint, after which we will use the average execution time for our final comparison.

4.3.1 Experimental environment

Both SMTreeSolver and treeSolver are implemented in Ruby. Therefore, we will use Ruby's built-in Time class to measure the time it takes to solve the given constraints. The experiments are run on a machine running Ubuntu 18.04 with an Intel i7-6700HQ 8-core CPU and 16 GB RAM.

[1] `https://github.com/taw/z3`

```ruby
# Create a new solver instance and an instance for
# creating expressions.
@solver = Smtreesolver::Solver.new
@expr = Smtreesolver::Expr.new

# Define the variables that need to be solved.
x = Smtreesolver::IntVar.new('X')
y = Smtreesolver::IntVar.new('Y')

# Define the constraint constants.
ten = Smtreesolver::Int.new(10)
zero = Smtreesolver::Int.new(0)

# Define the expressions that make up the constraints.
plus = @expr.add([x.ast_node, y.ast_node])
eq = @expr.eq(plus, ten.ast_node)

ge1 = @expr.ge(x.ast_node, zero.ast_node)
ge2 = @expr.ge(y.ast_node, zero.ast_node)

# Send the variables and expressions to the solver,
# along with the desired valuation method and print
# the results.
answer = @solver.solve([x, y], [eq, ge1, ge2], :min)
puts answer
```

```
Output:
{"X" => 0, "Y" => 10}
```

Figure 4.2: Ruby implementation of the constraint given in Figure 3.2.

Table 4.1: Types of constraints used in the performance comparison between SMTreeSolver and treeSolver.

Category	**Description**
Sparse domains	Constraints over integers and reals that contain 'holes', e.g. are of the form $a < x < b \wedge c < x < d$, where x is a variable and a, b, c, and d are constants, with $a < b < c < d$.
Variable addition	Arithmetic addition constraints over integers, e.g. constraints of the form $x + y = c$, where x and y are variables and $0 < c < 5000$ is a constant.
Combined constraints	A set of constraints that combines different integers, real numbers and strings. This set consists of the following constraints: $i_1 > c_{min} \wedge i_1 < c_{max}$ $i_2 \geq c_1 + 1 \wedge i_2 \leq c_{max}$ $r_1 > c_{min} \wedge r_1 < c_{max}$ $r_2 \geq c_2 + 1 \wedge r_2 \leq c_{max}$ $s = c_3$ Where i_1, i_2 are integer variables, r_1, r_2 are real variables, s is a string variable, c_1, c_2, c_3 are randomly generated constants, $c_{min} = 0$ and $c_{max} = 268435455$ (the largest value possible in GNU Prolog).

CHAPTER 5

Results

In this section, we present the results of our implementation. We will evaluate the suitability of using Z3 in a model-based testing environment (in this case, the Axini Modeling Suite) using the requirements specified in the previous chapter, and we compare our implementation to an already-existing constraint solver for test case generation using GNU Prolog.

5.1 Support for different valuation methods

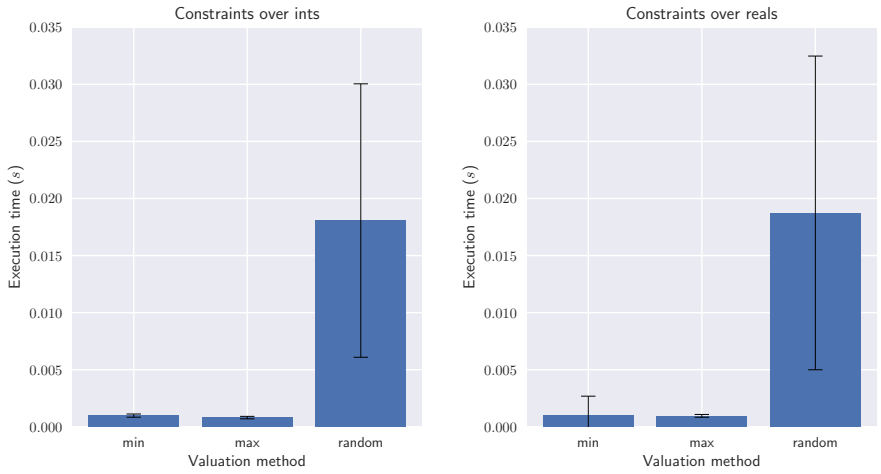

Figure 5.1: Average execution time over 500 runs for constraints of the form $a < x < b \wedge c < x < d$, where x is a variable and a, b, c, and d are randomly selected constants between 0 and 5000, with $a < b < c < d$.

Because of its built-in optimization feature, Z3 is able to find the boundary value of a variable within a reasonable time. However, as can be seen in Figure 5.1, the proposed method of finding random values takes around 20 times as long compared to the boundary variables. Furthermore, there is a significant discrepancy between the time it takes to find a random solution for different constraints, which for sparse domains is illustrated in Table 5.1. In fact, in some cases (e.g. on constraints with dependent variables over real numbers) the proposed method for finding takes minutes or longer, even for simple constraints. The reason behind this is that our method does not take into account these dependencies. This could be solved by detecting which variables depend on each other and only assigning a random solution to one of these variables and letting the solver pick solutions for the remaining variables using this random assignment. However, to be able to do so, we would need to be able to recognize which variables depend on each other, which is not possible in the current implementation of SMTreeSolver.

Table 5.1: Execution times (average, minimum, and maximum) for constraints over sparse domains.

Variable type	**Valuation method**	**Average**	**Standard deviation**	**Minimum**	**Maximum**
Integer	min	$9.811 \cdot 10^{-4}$	$1.394 \cdot 10^{-4}$	$8.584 \cdot 10^{-4}$	$2.714 \cdot 10^{-3}$
	max	$8.063 \cdot 10^{-4}$	$1.081 \cdot 10^{-4}$	$7.082 \cdot 10^{-4}$	$1.319 \cdot 10^{-3}$
	random	$1.807 \cdot 10^{-2}$	$1.198 \cdot 10^{-2}$	$1.115 \cdot 10^{-2}$	$9.653 \cdot 10^{-2}$
Float	min	$1.007 \cdot 10^{-3}$	$1.676 \cdot 10^{-3}$	$8.152 \cdot 10^{-4}$	$3.843 \cdot 10^{-2}$
	max	$9.586 \cdot 10^{-4}$	$1.221 \cdot 10^{-4}$	$8.340 \cdot 10^{-4}$	$1.916 \cdot 10^{-3}$
	random	$1.874 \cdot 10^{-2}$	$1.374 \cdot 10^{-2}$	$1.080 \cdot 10^{-2}$	$1.151 \cdot 10^{-1}$

5.2 Support for different data types

As noted in Chapter 3.3.2, Z3 supports different data types and allows users to define their own data types. Currently, SMTreeSolver supports boolean values, integers and reals, and limited support for strings. Whereas the Prolog-based treeSolver requires an encoder that encodes incoming values into a finite (integer) domain, constraints in SMTreeSolver can largely be evaluated 'as-is'. This solves the problem treeSolver currently has where certain values have to be within a certain range to prevent their encodings from exceeding GNU Prolog's domain. Moreover, more complex constraints are possible by using Z3, because the restrictions that encoding the data types introduced are taken away.

Whereas in GNU Prolog (and thus in treeSolver), a set of constraints is evaluated at once by joining the individual constraints by logical 'and'-expressions, Z3 (and therefore SMTreeSolver) allows for independent assertions as well as joined assertions. We compared these two methods of evaluating by using the constraint combining different data types as defined in Table 4.1 and solving it using SMTreeSolver, once by asserting sub-constraints over different data types independently and once by asserting it as one big constraint. The results are shown in Figure 5.2. It is clear that asserting sub-constraints over different types independently reduces the time it takes to solve a constraint by approximately 50%, which is a significant difference. A reason for this could be that by asserting constraints over different data types independently, Z3 is able to optimize the solver better for each sub-constraint.

5.3 Performance compared to treeSolver

One of the requirements of SMTreeSolver is that it should be able to solve the given constraints within a reasonable time. Ideally, it should match or exceed the performance of Axini's current treeSolver implementation, but it should at least be able to solve commonly used constraints within a tenth a of second. Figure 5.3 shows that although SMTreeSolver meets the latter requirement, it is significantly slower than the treeSolver. Furthermore, depending on the type of constraint, there seems to a larger difference in performance between the minimum and maximum valuation methods with SMTreeSolver. This is less apparent in the treeSolver. For both solvers, there is a significant deviation in the performance which applies to all types of constraints that have been analyzed. This can be explained by the fact that the constant values are randomly generated, where constraints with a larger 'gap' between different values generally take longer to be solved.

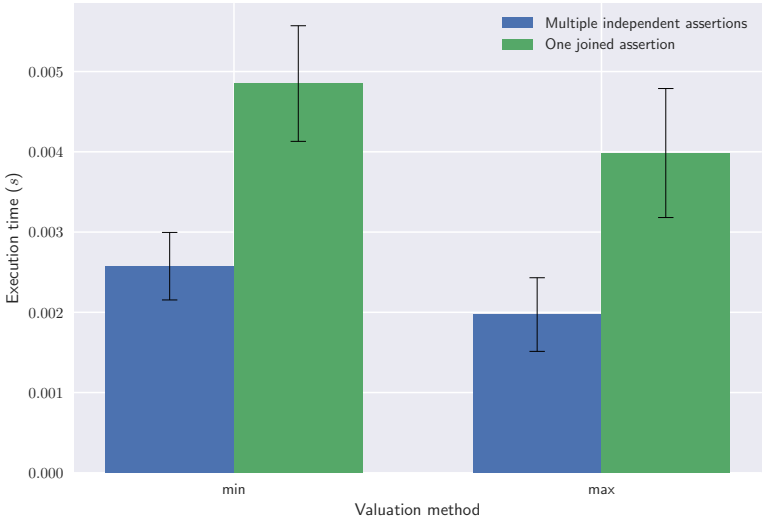

Figure 5.2: Comparison of performance when asserting each constraint independently versus as one constraint joined by logical 'and'-expressions (average over 500 runs).

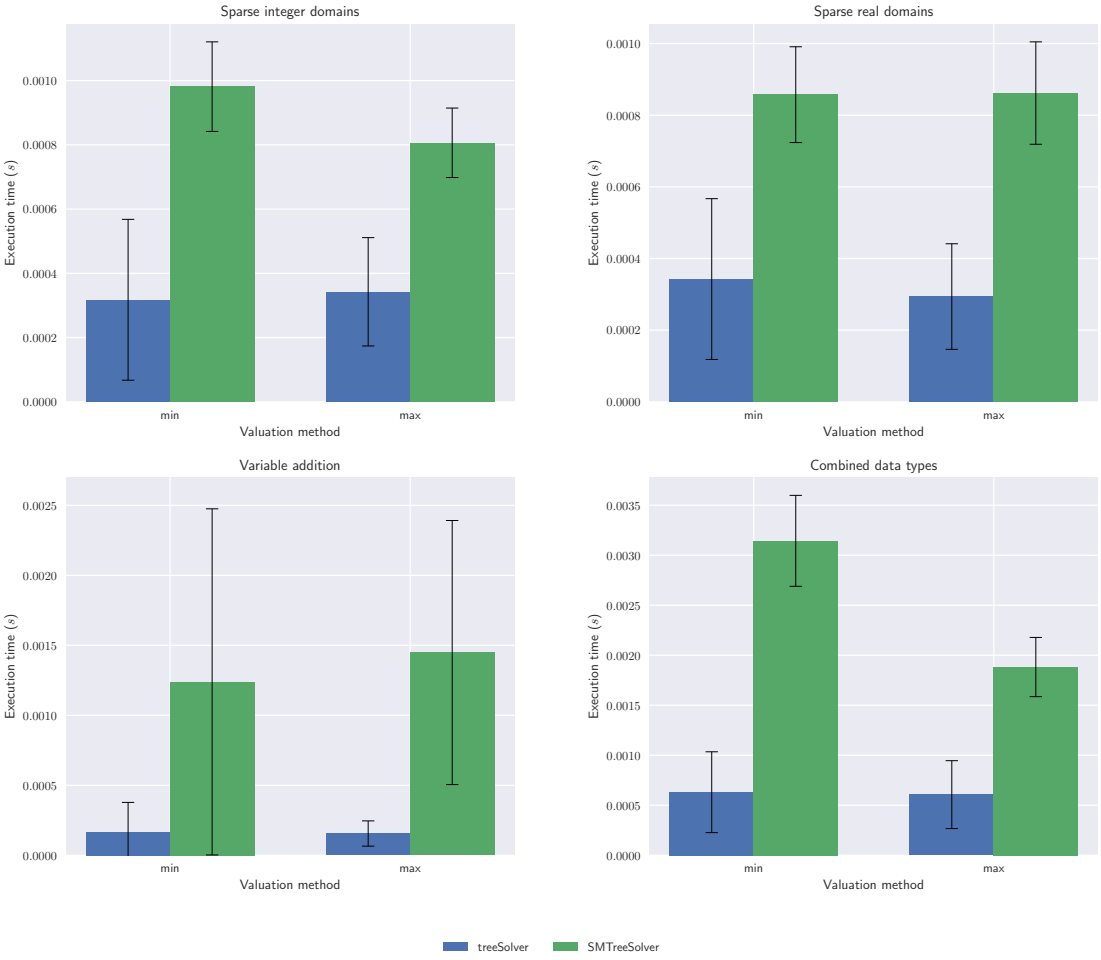

Figure 5.3: Performance of the treeSolver implementation using Prolog compared to the SMTreeSolver implementation using Z3 (average over 500 runs, using the constraints given in Table 4.1).

CHAPTER 6

Discussion

This chapter will discuss the research questions presented in the introduction of this thesis and will be answered based on our findings during our research. Furthermore, we will address some threats that could reduce the validity of our research and discuss how we attempted to minimize these threats.

6.1 Revisiting the research question

By composing the following research questions, we aimed to answer the central question of this thesis, *how can we use SMT solvers in a model-based testing context?*:

1. How is constraint solving used in model-based testing?
2. What is the state-of-the-art in SMT solvers?
3. How can we use SMT solvers to generate test case data?
4. How can we implement an SMT solver in an existing model-based testing tool?
5. How does an SMT solver compare to other types of constraint solvers?

In this section, we will discuss each question and our corresponding findings separately.

6.1.1 How is constraint solving used in model-based testing?

The model-based testing approach that is central in this thesis is by using Symbolic Transition Systems (STS) that describe the behavior of a system. Transitions can contain constraints that depend on location variables and interaction parameters that determine whether a constraint can take place or not. Test cases are generated by selecting a set of sequences from an STS (a *trace*) and using the corresponding constraints to determine whether a trace can be traversed in the first place and to generate test case data. This is done with a constraint solver. Two types of constraint solvers that are typically used for this purpose are constraint solvers using Constraint Logic Programming or SMT solvers. While both types of solvers ultimately achieve the same objective, namely validating a set of constraints and generating a set of solutions, the approach to achieving this differs.

6.1.2 What is the state-of-the-art in SMT solvers?

SMT solvers have improved significantly in recent years. Where classical approaches to SMT solving essentially encompassed translating constraints to boolean formulas and solving them using a SAT solver, modern SMT solvers offer a set of different approaches for different types of constraints. Z3, one of the most versatile SMT solvers currently available, is able to solve numerical constraints as well as constraints consisting of strings, arrays, and user-defined data

types. Furthermore, by using its optimization features it is possible to obtain the boundary values of a constraint variable.

6.1.3 How can we use SMT solvers to generate test case data?

To generate test case data, the constraint solver needs to be able to find the boundary values of a variable for which the given set of constraints holds, as well as be able to generate random solutions. As pointed out in the discussion of the previous research question, we can use Z3's optimization features to find these boundary values of a variable. Z3 does not offer a built-in possibility for generating random solutions, but in some cases, we can use the boundary values to generate a set of random solutions that satisfy the given constraints.

6.1.4 How can we implement an SMT solver in an existing model-based testing tool?

For this thesis, we were able to use the Axini Modeling Suite (AMS). This is a model-based testing tool that uses Symbolic Transition Systems to generate test cases and GNU Prolog as its constraint solver. The Axini Modeling Suite is implemented in Ruby. Because of this, we implemented our interface to Z3, dubbed 'SMTreeSolver' after the already existing 'treeSolver', in Ruby as well. Since Z3 itself does not provide bindings to Ruby, we used the Ruby-FFI library to create bindings to the native C API. The constraints in AMS are represented as abstract syntax trees, which can be used as input to the SMTreeSolver interface which will use Z3 to obtain the requested values.

6.1.5 How does an SMT solver compare to other types of constraint solvers?

We compared our SMTreeSolver implementation using Z3 to the already existing treeSolver implementation that uses the finite domain solver GNU Prolog. While we found that using Z3 instead of a finite domain solver is more suitable to handle constraints over different data types, since most of these data types are natively supported by Z3, we did find that that there is a decrease in performance compared to a Prolog-based solver. Nevertheless, the performance of SMTreeSolver can still be considered 'reasonable', as it is capable of solving the constraints used in our experiments within a hundredth of a second. Furthermore, the proposed solution for finding random values that satisfy a given constraint proved to be insufficient for our purposes, especially when there are dependencies between variables in a constraint.

6.2 Threats to validity

The largest threat to the validity of our findings is that, due to the scope of this research, we have not yet been able to test our solution in a 'real-world' environment, e.g. by integrating it into the Axini Modeling Suite and validating it against a verified model. Because the intent is to apply our solution in the Axini Modeling Suite, we still have to do this outside of this thesis, along with writing an elaborate set of tests in general to ensure the robustness of our implementation.

As for the validity of our experiments, we aimed to provide our results as reliably as possible by repeating each of our experiments a sufficient amount of times. The results presented are unfiltered, which may cause outliers to skew some of the results. However, we believe that because of the nature of our research even the outliers are of importance because these can appear in real-world scenarios as well.

Lastly, our aim was to make our experimental set-up for comparing the SMTreeSolver against the treeSolver as identical as possible in an attempt to eliminate any outside factors that could influence the results of the measurements. This was primarily done by using the programming language Ruby for both implementations and using the same method of measuring performance. However, it is still possible that during the execution of our experiments, some (background) processes were using parts of the available resources. While we deem the chances of this influencing our final results to be minimal, a repetition of our experiments on a dedicated server or a virtual machine with dedicated resources might confirm that our measurements are reliable.

CHAPTER 7

Conclusions

The central question of this thesis was

How can we use SMT solvers in a model-based testing context?

We answered this question by first providing a background to model-based testing using Symbolic Transition Systems and by examining how constraint solvers, in general, are used in this context. Using this background, we compiled a set of requirements and features that an SMT solver should support to be usable for model-based testing. We found that the SMT solver Z3 best suits these requirements. We then proceeded to create an interface (SMTreeSolver) to Z3 that can be used in an already existing model-based testing tool, the Axini Modeling Suite (AMS) to be precise. Because AMS already employs a constraint solver using GNU Prolog, our implementation would ideally match or best the performance of this solver. While our implementation can solve given constraints within a reasonable time (e.g. within a hundredth of a second), it is not able to match the performance of the currently implemented solver. Since our implementation is able to support different data types and structures without the intervention of an encoder like the current implementation requires, more possibilities regarding the types of constraints to be solved are introduced. For example, it would be possible to solve constraints over strings using regular expressions or constraints over user-defined data types. Having these possibilities shows that the use of SMT solvers is a valuable addition to the model-based testing landscape.

7.1 Further Work

While our research shows that SMT solvers are suitable to be used in model-based testing, there are still some steps to be made before our implementation can be used in an industrial setting like AMS. Assuming SMTreeSolver will eventually be used within AMS, we first of all need to ensure that all data types the Axini Modeling Language supports are also supported by SMTreeSolver. While most data types can be directly implemented because Z3 offers native support for constraints with these data types, there some types that Z3 doesn't immediately recognize. One solution would be to use Z3's support for user-defined data types. However, it should be examined whether that would be the preferable solution over the encoding method Axini currently uses for its Prolog-based solver.

In terms of real-world usability, the method proposed in this thesis for finding random solutions has proven to be insufficient, especially for constraints where dependencies between variables appear. We already touched upon possible solutions to this in Chapter 5.1, but an in-depth study into finding random solutions to a given constraint could further enhance the use of SMT solvers in model-based testing.

Another point of attention is the memory management of the SMTreeSolver. Because its implementation in Ruby is bound directly to Z3's native C API, we have to be extra careful in terms of memory management. Examining the memory usage and especially potential memory leaks was outside the scope of this research, should be tested thoroughly before being deployed to a production environment.

Lastly, we mentioned in Chapter 3.3.2 that Z3 (and SMT solvers in general) do not yet offer support for nonlinear arithmetic, except nonlinear polynomial real arithmetic. Solving this issue is far beyond trivial, and even impossible for integer arithmetic. However, for the specific purpose of generating test cases, it might be possible to create a 'best-effort' algorithm that generates a set of constraints that approximates the required test case data.

Bibliography

[1] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta and D. Kroening. 2010.

[2] Clark Barrett et al. "CVC4". In: *Computer Aided Verification*. Springer Berlin Heidelberg, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007%2F978-3-642-22110-1_14.

[3] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "νZ - An Optimizing SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2015, pp. 194–199. DOI: 10.1007/978-3-662-46681-0_14. URL: https://doi.org/10.1007%2F978-3-662-46681-0_14.

[4] Andrei A. Bulatov, Andrei A. Krokhin, and Peter Jeavons. "Constraint Satisfaction Problems and Finite Algebras". In: *Automata, Languages and Programming*. Springer Berlin Heidelberg, 2000, pp. 272–282. DOI: 10.1007/3-540-45022-x_24. URL: https://doi.org/10.1007%2F3-540-45022-x_24.

[5] T.S. Chow. "Testing Software Design Modeled by Finite-State Machines". In: *IEEE Transactions on Software Engineering* SE-4.3 (May 1978), pp. 178–187. DOI: 10.1109/tse.1978.231496. URL: https://doi.org/10.1109%2Ftse.1978.231496.

[6] Alessandro Cimatti et al. "Experimenting on Solving Nonlinear Integer Arithmetic with Incremental Linearization". In: *Theory and Applications of Satisfiability Testing – SAT 2018*. Springer International Publishing, 2018, pp. 383–398. DOI: 10.1007/978-3-319-94144-8_23. URL: https://doi.org/10.1007%2F978-3-319-94144-8_23.

[7] Alessandro Cimatti et al. "The MathSAT 5 SMT Solver ". In: 2012.

[8] Edmund M. Clarke et al. "Model Checking and the State Explosion Problem". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 1–30. DOI: 10.1007/978-3-642-35746-6_1. URL: https://doi.org/10.1007%2F978-3-642-35746-6_1.

[9] Philippe Codognet and Daniel Diaz. "Compiling constraints in clp(FD)". In: *The Journal of Logic Programming* 27.3 (June 1996), pp. 185–226. DOI: 10.1016/0743-1066(95)00121-2. URL: https://doi.org/10.1016%2F0743-1066%2895%2900121-2.

[10] Daniel Diaz and Philippe Codognet. "The GNU Prolog system and its implementation". In: *Proceedings of the 2000 ACM symposium on Applied computing - SAC '00*. ACM Press, 2000. DOI: 10.1145/338407.338553. URL: https://doi.org/10.1145%2F338407.338553.

[11] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification*. Springer International Publishing, 2014, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49. URL: https://doi.org/10.1007%2F978-3-319-08867-9_49.

[12] L. Frantzen, J. Tretmans, and T. A. C. Willemse. "A Symbolic Framework for Model-Based Testing". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 40–54. DOI: 10.1007/11940197_3. URL: https://doi.org/10.1007%2F11940197_3.

[13] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. "Test Generation Based on Symbolic Specifications". In: *Formal Approaches to Software Testing*. Springer Berlin Heidelberg, 2005, pp. 1–15. DOI: 10.1007/978-3-540-31848-4_1. URL: https://doi.org/10.1007%2F978-3-540-31848-4_1.

[14] Ardavan Ghaffari. "Trace Coverage Strategy for Symbolic Transition Systems". MA thesis. Amsterdam: University of Amsterdam, Aug. 2016.

[15] Pascal Van Hentenryck, Helmut Simonis, and Mehmet Dincbas. "Constraint satisfaction using constraint logic programming". In: *Artificial Intelligence* 58.1-3 (Dec. 1992), pp. 113–159. DOI: 10.1016/0004-3702(92)90006-j. URL: https://doi.org/10.1016%2F0004-3702%2892%2990006-j.

[16] W.E. Howden. "Methodology for the Generation of Program Test Data". In: *IEEE Transactions on Computers* C-24.5 (May 1975), pp. 554–560. DOI: 10.1109/t-c.1975.224259. URL: https://doi.org/10.1109%2Ft-c.1975.224259.

[17] Joxan Jaffar and Michael J. Maher. "Constraint logic programming: a survey". In: *The Journal of Logic Programming* 19-20 (May 1994), pp. 503–581. DOI: 10.1016/0743-1066(94)90033-7. URL: https://doi.org/10.1016%2F0743-1066%2894%2990033-7.

[18] Dejan Jovanović and Leonardo de Moura. "Solving Non-linear Arithmetic". In: *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*. Vol. 7364. Lecture Notes in Computer Science. Springer, 2012, pp. 339–354.

[19] Vipin Kumar. "Algorithms for Constraint-Satisfaction Problems: A Survey". In: *AI Magazine* 13 (1992), pp. 32–44. DOI: 10.1609/aimag.v13i1.976. URL: https://doi.org/10.1609/aimag.v13i1.976.

[20] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories". In: *Communications of the ACM* 54.9 (Sept. 2011), pp. 69–77. DOI: 10.1145/1995376.1995394. URL: https://doi.org/10.1145%2F1995376.1995394.

[21] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007%2F978-3-540-78800-3_24.

[22] Rajesh Natarajan et al. "Intelligent automation". In: *World Quality Report 2019-20* (Oct. 2019), pp. 16–19. URL: https://www.sogeti.com/explore/reports/world-quality-report-2019/.

[23] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT Modulo Theories". In: *Journal of the ACM* 53.6 (Nov. 2006), pp. 937–977. DOI: 10.1145/1217856.1217859. URL: https://doi.org/10.1145%2F1217856.1217859.

[24] Jorge Mora Perdiguero. "Development of an Efficient Data Coverage Strategy for TestManager". MA thesis. Amsterdam: Vrije Universiteit Amsterdam, 2016.

[25] Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. "Symbolic execution of programs with strings". In: *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference on - SAICSIT '12*. ACM Press, 2012. DOI: 10.1145/2389836.2389853. URL: https://doi.org/10.1145%2F2389836.2389853.

[26] S.C. Reid. "An empirical analysis of equivalence partitioning, boundary value analysis and random testing". In: *Proceedings Fourth International Software Metrics Symposium*. IEEE Comput. Soc. DOI: 10.1109/metric.1997.637166. URL: https://doi.org/10.1109%2Fmetric.1997.637166.

[27] *Ruby FFI*. https://github.com/ffi/ffi. Accessed: 2020-05-04.

[28] *Ruby Programming Language*. https://www.ruby-lang.org/en/. Accessed: 2020-05-04.

[29] Roberto Sebastiani and Patrick Trentin. "On Optimization Modulo Theories, MaxSMT and Sorting Networks". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2017, pp. 231–248. DOI: 10.1007/978-3-662-54580-5_14. URL: https://doi.org/10.1007%2F978-3-662-54580-5_14.

[30] Roberto Sebastiani and Patrick Trentin. "OptiMathSAT: A Tool for Optimization Modulo Theories". In: *Journal of Automated Reasoning* 64.3 (Dec. 2018), pp. 423–460. DOI: 10.1007/s10817-018-09508-6. URL: https://doi.org/10.1007%2Fs10817-018-09508-6.

[31] Jan Tretmans. "Model Based Testing with Labelled Transition Systems". In: *Formal Methods and Testing*. Springer Berlin Heidelberg, 2008, pp. 1–38. DOI: 10.1007/978-3-540-78917-8_1. URL: https://doi.org/10.1007%2F978-3-540-78917-8_1.

[32] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches". In: *Software Testing, Verification and Reliability* 22.5 (Apr. 2011), pp. 297–312. DOI: 10.1002/stvr.456. URL: https://doi.org/10.1002%2Fstvr.456.

[33] Margus Veanes et al. "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer". In: *Formal Methods and Testing*. Springer Berlin Heidelberg, pp. 39–76. DOI: 10.1007/978-3-540-78917-8_2. URL: https://doi.org/10.1007%2F978-3-540-78917-8_2.

[34] *Z3 C API*. https://z3prover.github.io/api/html/group__capi.html. Accessed: 2020-05-18.

[35] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. "Z3-str: a z3-based string solver for web application analysis". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, 2013. DOI: 10.1145/2491411.2491456. URL: https://doi.org/10.1145%2F2491411.2491456.