Model Checking of Symbolic Transition Systems with SMT Solvers

We investigate the feasibility of using SMT solvers for Model Checking. Specifically, we consider reachability properties for Symbolic Transition Systems (STSs). SMT solvers are a generalisation of SAT solvers that determine the logical satisfiability of formulas in more advanced theories than only propositional logic. STSs are transition systems providing a finite representation to model the state behaviour of systems with possibly infinite state spaces.

We consider three different approaches to the Model Checking problem with a varying degree of SMT solver involvement: first, an explicit approach without using SMT solvers; second, an approach using only an SMT solver; and third, a hybrid approach employing an external algorithm which uses SMT solvers for part of the problem.

As expected, the explicit approach is not powerful enough to handle infinite state spaces of STSs. Furthermore, we demonstrate that using only an SMT solver for Model Checking STSs stretches the theoretical and practical capabilities of SMT solvers. Finally, we show that the hybrid approach provides the highest degree of control over the search, yielding the best approach to handle the infinite state space of STSs. For the hybrid approach we present a prototype tool. Furthermore, we describe mathematically proven reductions in size of the generated constraints: once from quadratic to linear and once by a factor of two, compared to previous research.

Our conclusion is that using SMT solvers in hybrid Model Checking approaches provides a powerful and promising approach to improve model verification of infinite state space systems.

Master Thesis

Carsten Rütz
8th of November 2013
Radboud University Nijmegen

Supervisors

Dr. Ir. G.J. Tretmans
Dr. Ir. M. van der Bijl
Prof. Dr. J. van Eijck
Prof. Dr. F.W. Vaandrager

Contents

Acknowledgements

I would like to thank the following people for their effort in supporting me and my work on this thesis: Jan Tretmans, especially for the many hours of discussion – six hours on a day were not an exception – and his precise and detailed feedback; Machiel van der Bijl and Menno Jonkers, owners of Axini, especially for their insights from the practice of software engineering and their moral support; Jan van Eijck, in particular for his help on the mathematical proofs; and finally, Frits Vaandrager for being the second reader of this thesis.

1. Introduction

Nowadays, computer-supported systems are present everywhere. The importance of their correctness varies depending on the context they are used in. While malfunctioning systems may cause loss of business, they may even endanger the safety of people in some cases. Mobile phones, for example, may be annoying when not working flawlessly. Malfunctioning airplanes and medical machinery, however, may cause loss of lives.

Testing is a way to increase confidence in the correctness of systems. Model-Based Testing (MBT) is a testing technique that uses formal descriptions of systems to automatically derive test cases [NSVT07, UPL11]. These formal descriptions are usually state graphs with transitions between states describing the logical flow of the system; they are also known as transition systems. The research reported in this document is conducted in cooperation with Axini, a company applying MBT to medical, financial and transporting (embedded) systems. As formalism to describe systems, Axini uses a special kind of transition systems called Symbolic Transition Systems (STSs) [FTW06]. STSs enable Axini to use rich data representations ranging from integers, to lists and abstract data types to describe real-life systems.

Formal descriptions of systems often grow big and, therefore, become difficult to interpret for humans. This can be improved by reasoning about such formal descriptions. Proving properties automatically is a form of such reasoning. For Axini, proving properties of their formal descriptions can increase the quality of testing in at least two ways. First, the correctness of the formal description itself, i.e. the specification of the system under test, can be verified. Second, guaranteeing the invariance of valid properties after changes to the formal description helps maintaining the correctness of the formal description throughout the testing process. The latter is thus an implementation of regression testing [Mye04] of the specification.

We chose Model Checking (MC) as a technique to prove properties of such formal descriptions of systems. Model Checking is a technique to determine whether a property expressed in a temporal logic holds. We built a model checker for STSs in which we used Satisfiability Modulo Theories (SMT) solvers as a basis. SMT solvers are able to check the satisfiability of logical formulae [DMB11]. By translating the Model Checking problem into a logical formula, these SMT solvers can be exploited for Model Checking [GRV09, AMP06]. Reasons to choose SMT solvers as a basis for our model checker include their support for a wide range of data types in formulae and the relative novelty of their application to the Model Checking problem.

We describe three different approaches to the Model Checking problem using varying degrees of SMT solver involvement: first, a traditional explicit state model checker using no SMT solver; second, a model checker encoding the whole Model Checking problem into an SMT solver constraint; and third, a hybrid solution consisting of an algorithm using the SMT solver for parts of the problem. We show that it is feasible to develop a model checker for reachability properties including all components of STSs and the data types boolean and integer. Furthermore, we show that the hybrid approach is most promising. Our high-level syntactical reasoning with STSs includes infinite data structures and infinite state spaces. Therefore, our approach differs from traditional explicit state and symbolic model checking which reason semantically about finite data structures and/or employs bounded searches.

In the following, we briefly describe our research question and the methodology (section 2). Then, we present the theoretical background necessary for this thesis (section 3). Furthermore, we describe our three approaches (section 4) and discuss them (section 5). Finally, we summarise our findings (section 6), draw conclusions and suggest future work (section 7).

2. Research Question and Methodology

Our research question was: *How can a Model Checker for Symbolic Transition Systems (STSs) be built based on Satisfiability Modulo Theories (SMT)?*.

We broke this main question down into the following sub-questions:

- How can the general structure (locations and switches) of an STS be represented in a model checker based on SMT?
- Should the whole STS be represented as a single constraint problem?

- Should only constraints and their free variables be solved by the SMT solvers?
- What data (types) can be handled by SMT solvers (or SMT standards)?
- What techniques of SMT solvers can be exploited for the use in a model checker?
- How can the data types used in STSs be translated for SMT solvers?
- How can the other components (updates, gates, restrictions) of an STS be represented in an SMT model checker?
- How can free variables in these components be handled?

We used a constructive proof as method to derive an answer to the research question: To see how a model checker for STSs based on SMT can be built, we built one. In our approach to developing such a model checker, we divided the desired functionality into parts and developed each part in a separate phase. This resembles an agile software development approach, which was appropriate for our problem because feasibility results from one phase were needed in subsequent phases.

3. Theoretical Background

We present the theoretical background that is relevant to this thesis in four parts. First, we give a short summary of first-order logic with its concepts and the notation we use. Second, we describe the different approaches to Model Checking and previous research in this field. Third, we introduce Satisfiability Modulo Theories (SMT) and present previous research on using SMT for Model Checking. Fourth, we define Symbolic Transition Systems (STSs), our input formalism along with its semantics.

3.1. First-Order Logic

We give a brief description of specific concepts and notations in first-order logic that we use in this thesis. If not noted otherwise, these descriptions are literally taken from "A Symbolic Framework for Model-Based Testing" by Frantzen et al. [FTW06]. For an extensive introduction, see "A Concise Introduction to Mathematical Logic" by Rautenberg [Rau06], for example.

For sets A and B, the set of all total functions from A to B is denoted B^A.

We consider a first-order structure $(\mathfrak{S}, \mathfrak{M})$. Then, $\mathfrak{S} = (F, P)$ is a logical signature where F is the set of *function symbols*. Each $f \in F$ is assigned an arity $n \in \mathbb{N}$. Furthermore, P is a set of *predicate symbols*. Each $p \in P$ is also assigned an arity $n \in \mathbb{N}^+$. The model is $\mathfrak{M} = (\mathfrak{U}, (f_\mathfrak{M})_{f \in F}, (p_\mathfrak{M})_{p \in P})$ where $\mathfrak{U}$ is a non-empty set called *universe*. For all $f \in F$ with arity n, $f_\mathfrak{M}$ is a function of type $\mathfrak{U}^n \to \mathfrak{U}$. For all $p \in P$ with arity n, $p_\mathfrak{M} \subseteq \mathfrak{U}^n$.

Let $\mathfrak{X}$ be a set of *variables* and $X, Y \subseteq \mathfrak{X}$. *Terms* over X, denoted $\mathfrak{T}(X)$, are built of variables $x \in X$ and function symbols $f \in F$. The set of all variables in a term t is denoted $var(t)$. A *term-mapping* is a function $\sigma : \mathfrak{X} \to \mathfrak{T}(\mathfrak{X})$. Furthermore, a term-mapping σ applied to a tuple $\langle x_1, \ldots, x_n \rangle$ is defined as $\sigma(\langle x_1, \ldots, x_n \rangle) = \langle \sigma(x_1), \ldots, \sigma(x_n) \rangle$. The identity term-mapping id is defined as $id(x) = x$ for all $x \in \mathfrak{X}$. A restricted term-mapping σ_X is equal to the term-mapping σ but only applied to variables in X, i.e. $\sigma_x(x) = \sigma(x)$ if $x \in X$ and x otherwise. The set of all term-mappings $\sigma \in \mathfrak{T}(\mathfrak{X})^\mathfrak{X}$ with $\sigma(x) \in \mathfrak{T}(Y)$ for all $x \in X$ and $\sigma(x) = x$ for all $x \notin X$ is denoted $\mathfrak{T}(Y)^{\mathfrak{X}]X}$. We omit the mentioning of $\mathfrak{X}$ and write $\mathfrak{T}(Y)^X$ instead.

The set of *free variables* of a first-order formula φ is denoted $\text{free}(\varphi)$. The set $\mathfrak{F}(X)$ denotes all first-order formulas φ that satisfy $\text{free}(\varphi) \subseteq X$.

A *valuation* is a function $\vartheta \in \mathfrak{U}^\mathfrak{X}$. A valuation applied to a tuple of variables $\langle x_1, \ldots, x_n \rangle$ is defined as $\vartheta(\langle x_1, \ldots, x_n \rangle) = \langle \vartheta(x_1), \ldots, \vartheta(x_n) \rangle$. The evaluation of terms based on such a valuation ϑ, called a *term-evaluation*, assigns elements of the universe $\mathfrak{U}$ to terms and is denoted $\vartheta_{eval} : \mathfrak{T}(\mathfrak{X}) \to \mathfrak{U}$.

Let ϕ and ψ be first-order formulas. Then, ϕ is *satisfiable* iff there is a model M under which ϕ evaluates to true: $\exists M : M \models \phi$, also noted $SAT(\phi)$. ϕ is *logically valid* iff it is true under every model M: $\forall M : M \models \phi$, or short $\models \phi$. Furthermore, ψ is a *logical consequence* of formula ϕ iff for every model M under which ϕ evaluates to true also ψ evaluates to true: $\forall M : M \models \phi \implies M \models \psi$. Finally, ϕ and ψ are *equivalent* $\phi \equiv \psi$ iff $\forall M : M \models \phi \Leftrightarrow M \models \psi$ ([Rau06]).

3.2. Model Checking

Model Checking is a technique to verify properties of formal system descriptions. Given a formal system description and a property that describes the system's specification, Model Checking automatically determines whether the property holds for the formal system description. For example, the formal system description may be given as a state transition system, while the properties are described in a temporal logic. We present a very brief historical overview of Model Checking, name the most important approaches to this technique, and describe a classification of properties.

Model Checking was invented more than 30 years ago by Emerson & Clarke and Queille & Sifakis independently [EC82, QS82] to automatically prove properties about formally described systems. In 2007, Clarke, Emerson and Sifakis received the Turing Award for their work.

The first algorithms for Model Checking are now known as *Explicit* Model Checking [Cla08, Eme08] which greatly suffer from the infamous state space explosion problem. SPIN [Hol97] is an example of an explicit model checker. Later, McMillan improved the representation of formal descriptions with binary decision diagrams. This is referred to as *Symbolic* Model Checking [McM92] because of its non-explicit representation of states and transitions between them. His improvement led to an increase in the number of states that could be handled. SMV [McM92] and LTSmin [BvdPW10] are example implementations.

In another approach, *SAT-based* Model Checking [BCC$^+$99], the Model Checking problem is translated into a satisfiability problem of propositional formulas. Here, finite encodings of the transition system and data are used, leading to Bounded Model Checking, i.e. properties are proven for computation paths of bounded length only. Furthermore, many techniques were invented to alleviate the state space explosion problem, for example, partial-order reduction and compositional descriptions of systems [Cla08].

The properties that are used with Model Checking can be classified into *safety-* and *liveness* properties [Lam77, BK08]. Safety properties state that "something bad will never happen" and they are used to describe unsafe states that a system must not reside in. Then, Model Checking is employed to determine whether these states can be reached. Hence, the term *reachability property* is often employed with safety properties as well. Liveness properties, on the other hand, state that "something good will eventually happen" and, therefore, are also called *progress properties*. They are used, for example, to show that a system (from any state) will always be able to reach a certain "good" state. Model Checking of liveness properties is considered more difficult than Model Checking of safety properties. This is because counter examples for safety properties can be found by finite system runs to a state that violates the property, while counter examples for liveness properties can only be found by infinite system runs. We limited our research to reachability properties.

3.3. Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is a generalisation of the satisfiability problem for propositional formulae to other (preferably) decidable theories [DMB11]. Such theories include the theories of integers, reals, lists, and algebraic data types, for example. SMT solvers try to find a satisfying valuation for a given formulae in these supported theories. Examples of such solvers are CVC4 [BCD$^+$11], Yices [DDM06] and Z3 [DMB08].

We give an example to describe the concepts of satisfiability and validity of formulas:

Example 3.1
Let $\phi := (x = 1 \vee x = 2 \implies x = 1)$ with $x \in \mathbb{N}$. Then ϕ is satisfiable with, e.g., $M := [x := 1]$ or $M := [x := 3]$. Note that also $\neg\phi$ is satisfiable: $\neg\phi \iff (x = 1 \vee x = 2) \wedge x \neq 1$ is satisfiable with $M := [x := 2]$. However, ϕ is not valid, e.g. with $M := [x := 2]$, $M \not\models \phi$.

Let $\psi := (x > 2 \implies x > 1)$ with $x \in \mathbb{N}$. ψ is valid and $\neg\psi \iff (x > 2 \wedge x \leq 1)$ is not satisfiable. Also, $x > 1$ is a logical consequence of $x > 2$: $\forall M : M \models x > 2 \implies M \models x > 1$

□

Concerning the satisfiability of formulas, we give some general rewriting rules and a general relation between the satisfiability and validity of formulas:

Lemma 3.2
Let ϕ and ψ be formulas. Then

1. $\text{SAT}(\phi \wedge \psi) \quad \Longrightarrow \quad \text{SAT}(\phi) \wedge \text{SAT}(\psi)$
2. $\text{SAT}(\phi \vee \psi) \quad \Longleftrightarrow \quad \text{SAT}(\phi) \vee \text{SAT}(\psi)$
3. $\text{SAT}(\phi \implies \psi) \quad \Longleftrightarrow \quad \text{SAT}(\neg\phi) \vee \text{SAT}(\psi)$
4. $\neg\,\text{SAT}(\neg\phi) \quad \Longleftrightarrow \quad \models \phi$

□

Note that lemma 3.2.1 does not hold in reverse. Take as a counterexample $\phi := x > 10$ and $\psi := x < 10$, with $x \in \mathbb{N}$. For a proof of lemma 3.2 see the appendix.

3.3.1. Model Checking with Satisfiability Modulo Theories

Several groups of researchers investigated the use of SMT solvers for Model Checking. We identified three lines of work: first, a backward-reachability approach making partial use of SMT solvers; second, an approach encoding the whole Model Checking problem into an SMT solver query; and third, an abstraction technique involving the use of SMT solvers for retrieving concrete counter examples after Model Checking an abstract version of the system. In the following, we shortly describe these three approaches.

Ghilardi, Alberti et al. proposed a backward-reachability procedure [GNRZ08, GR09, GRV09] that resembles the CTL Model Checking algorithm for reachability properties [BK08]. They implemented their work in the MCMT model checker. Alberti et al. [ABG$^+$12] extended this approach with lazy abstraction techniques and the handling of quantified formulae, resulting in the SAFARI model checker. Furthermore, Conchon et al. [CGK$^+$12] used the approach of Ghilardi et al. with their own SMT solver Alt-Ergo and parallelised the computation. This resulted in the Cubicle model checker.

Milicevic and Kugler [MK11] proposed a different approach. They formulate computation traces as lists on which the transition system and the desired property are described as constraints. Then, SMT solvers are used to solve this constraint yielding either the unsatisfiability of the formula or a valuation that satisfies it. Their approach implements a forward search through the possible computation traces.

Using SMT solvers in Counter Example Guided Abstraction Refinement (CEGAR) techniques is another approach that was studied. With CEGAR, an abstract version of the formal description of a system is used for Model Checking. A counter example for a property that was found in the abstract version still needs to be validated in the actual formal description. Here, SMT solvers are used to solve constraints. Armando et al. [Arm09] used this approach in their SMT-CMBC model checker. Furthermore, Cordeiro et al. [CFMS12] developed the ESBMC model checker based on this approach.

3.4. Symbolic Transition Systems

Model Checking uses formal descriptions of systems. A formalism for such formal descriptions is a *Symbolic Transition System (STS)*. In the following, we first define *Labelled Transition Systems (LTS)* which form the basis of more expressive STSs and are used to define the semantics of STSs. Then, we define the STS formalism and present examples of LTSs and STSs.

State-based systems can be formally described as *Labelled Transition Systems (LTS)*. These LTS define possible transitions between pairs of states [FTW06]:

Definition 3.3
A *Labelled Transition System (LTS)* is a tuple $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$ where S is a set of *states* and $s_0 \in S$ is the *initial state*. The set Σ is a set of observable *action labels*. The action label $\tau \notin \Sigma$ denotes an unobservable action; Σ_τ abbreviates the set $\Sigma \cup \{\tau\}$. The relation $\rightarrow \subseteq S \times \Sigma_\tau \times S$ is the *transition relation*; $s \xrightarrow{\mu} s'$ abbreviates $(s, \mu, s') \in \rightarrow$.

□

Symbolic Transition Systems (STSs) are a more complex formalism than LTSs and provide support for modelling state behaviour of data-intensive systems. Vertices and edges in STSs are called *locations* and *switches*, respectively. Modelling the control flow of data-intensive systems is possible due to switches which are equipped with a *restriction*, a *gate* and an *update* component. First, the restriction is a boolean expression that after evaluation determines whether a switch is enabled and thus can be "executed". Second, the gate is a label describing observable behaviour of a system; it is parameterised with a tuple

of *interaction variables*. Third, the update component allows to change the state after a switch has been taken by assigning new values to location variables. The *state* of an STS consists of the location it resides in and the valuation of all location variables. Furthermore, a *transition* of an STS is a switch with a concrete instance of the switch gate with instantiated interaction variables. This implies that a switch connecting two locations actually connects multiple states. With STSs, we consider the terminology "switch" and "location" as syntactical elements while "state" and "transition" are semantical elements.

Our definition of STSs is based on the definition of Frantzen et al. [FTW06]. The most important change to Frantzen's definition is the introduction of sorts to location- and interaction variables.

Definition 3.4
An STS is a tuple $\mathcal{S} = \langle L, l_0, \mathcal{V}, \mathcal{I}, \Lambda, \rightarrow \rangle$ where L is a finite set of locations and l_0 is the initial location. $\mathcal{V}$ is a finite set of sorted location variables and $\mathcal{I}$ is a finite set of sorted interaction variables; $\mathcal{V} \cap \mathcal{I} = \varnothing$ and we set $\text{Var} =_{def} \mathcal{V} \cup \mathcal{I}$. As a convention, we prefix interaction variables with an underscore, e.g. $\_x \in \mathcal{I}$. Λ is a finite set of gates; the constant $\tau \notin \Lambda$ denotes an unobservable gate; Λ_τ abbreviates $\Lambda \cup \{\tau\}$. The relation $\rightarrow \subseteq L \times \Lambda_\tau \times \mathfrak{F}(\text{Var}) \times \mathfrak{T}(\text{Var})^{\mathcal{V}} \times L$ is the switch relation; $l \xrightarrow{\lambda,\varphi,\rho} l'$ abbreviates $(l, \lambda, \varphi, \rho, l') \in \rightarrow$, where φ is the switch restriction and ρ the update mapping. We use the following functions and vocabulary:

1. $\mathbf{arity} : \Lambda_\tau \rightarrow \mathbb{N}$ is the arity function,
2. $\mathbf{type}(\lambda)$ yields a tuple of size $\mathbf{arity}(\lambda)$ of interaction variables for gate λ,
3. $\mathbf{sort} : \mathfrak{T} \rightarrow \mathbb{S}$ is the sort function,
4. $\mathfrak{T}$ is a term that is either a literal of some sort $\mathbb{S}$, a variable in Var of sort $\mathbb{S}$ or an operator OP applied to arguments. 0 is a literal of sort *Int*, for example.
5. $\mathcal{S}$ is well-defined iff
   a) $\mathbf{arity}(\tau) = 0$
   b) $\mathbf{type}(\lambda)$ yields distinct interaction variables
   c) $l \xrightarrow{\lambda,\varphi,\rho} l'$ implies $\mathbf{free}(\varphi) \subseteq \mathcal{V} \cup \mathbf{type}(\lambda)$, i.e. the only free variables in a switch restriction are location variables and interaction variables that belong to the gate
   d) $\rho \in \mathfrak{T}(\mathcal{V} \cup \mathbf{type}(\lambda))^{\mathcal{V}}$, i.e. only location variables can be updated with values of other location- and interaction variables
   e) $\mathbf{sort}(\varphi) = \mathit{Bool}$, i.e. the switch restriction is well-sorted
   f) $v \mapsto t \in \mathfrak{T}(\text{Var})^{\mathcal{V}}$ implies $\mathbf{sort}(v) = \mathbf{sort}(t)$, i.e. the update mapping is well-sorted if the right-hand side of the assignment has the same sort as the location variable that is being assigned
   g) $v \mapsto t \in \iota$ implies $\mathbf{sort}(v) = \mathbf{sort}(t)$, i.e. the initial valuation of location variables is well-sorted
6. $\mathcal{S}(\iota)$ is an initialised STS, with location variable initialisation $\iota \in \mathfrak{U}^{\mathcal{V}}$, which assigns all variables from $\mathcal{V}$ in l_0 to an element from the universe $\mathfrak{U}$.

□

Semantics of STSs

The semantics of an STS (definition 3.5) is given by Frantzen et al. [FTW06] in terms of an LTS:

Definition 3.5
Let $\mathcal{S} = \langle L, l_0, \mathcal{V}, \mathcal{I}, \Lambda, \rightarrow \rangle$ be an STS. Its interpretation $[\![\mathcal{S}]\!]_\iota$ in the context of $\iota \in \mathfrak{U}^{\mathcal{V}}$, is defined as $[\![S]\!]_\iota = \langle L \times \mathfrak{U}^{\mathcal{V}}, (l_0, \iota), \Sigma, \rightarrow \rangle$, where $\Sigma = \bigcup_{\lambda \in \Lambda}(\{\lambda\} \times \mathfrak{U}^{\mathbf{arity}(\lambda)})$ is the set of actions and $\rightarrow \subseteq (L \times \mathfrak{U}^{\mathcal{V}}) \times (\Sigma \cup \{\tau\}) \times (L \times \mathfrak{U}^{\mathcal{V}})$ is defined by the following rule:

$$\frac{l \xrightarrow{\lambda,\varphi,\rho} l' \qquad \varsigma \in \mathfrak{U}^{\mathbf{type}(\lambda)} \qquad \vartheta \cup \varsigma \models \varphi \qquad \vartheta' = (\vartheta \cup \varsigma)_{eval} \circ \rho}{(l,\vartheta) \xrightarrow{\lambda,\varsigma(\mathbf{type}(\lambda)))} (l',\vartheta')}$$

The semantics of an initialised STS $\mathcal{S}(\iota)$ is given by the LTS $[\![\mathcal{S}]\!]_\iota$.

□

The set of states, the set of actions, and the transition relation depend on the universe $\mathfrak{U}$. Hence, if that universe contains infinite data types, e.g. integers, those sets can become infinite, and possibly yield an infinite state space.

Example 3.6
A calculator to add, subtract, multiply and divide numbers is described in this example (figure 1). The four actions "add", "sub", "mult", and "div" change the value of an internal total (location variable i). Specifically, the add-action adds an arbitrary integer greater or equal to 0 to the internal total, the sub-action subtracts an integer greater or equal to 0 from the internal total, the mult-action multiplies the internal total by an integer, and the div-action divides the internal total by a non-zero integer. The division action is an integer division.

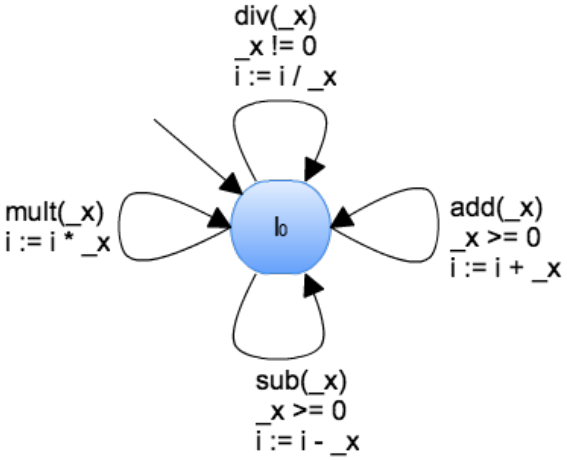
In terms of definition 3.4, l_0 is the initial location, i is the only location variable, and $\_x$ is the only interaction variable. There are four switches, one for each of the four actions. The switch for the add-action, for example, consists of the gate "add" with its interaction variable $\_x$, the restriction $\_x >= 0$ stating that only values greater or equal to zero may be added, and the update $i := i + \_x$. Note that the restriction is well-defined according to definition 3.4.5.c because it only involves the location variable i and the interaction variable $\_x$ belonging to the gate "add" that is used in the switch. Furthermore, the update is well-defined according to definition 3.4.5.d because the location variable i is updated with the values of a location variable – namely itself – and the interaction variable $\_x$ belonging to the involved gate "add".

While the syntactical description of the calculator is finite (figures 1a and 1b), its semantic is infinite (figures 1c and 1d): the infinite representation of integers yields an infinite state space $\{\langle l_0, \mathfrak{U}^{\mathcal{V}}\rangle\}$ as well as an infinite set of observable action labels Σ. We limited the semantical description to the add-action for clarity.

□

Example 3.7
One LTS and two STSs are described in our second example (figure 2). We will use these examples to compare the termination behaviour of different approaches to the Model Checking problem described below. First, the LTS example (figure 2a) describes a finite-state system in which one state is unreachable as there is no transition to it; the looping transition in state s_0 leads to possibly unbounded paths. Second, one STS example (figure 2b) depicts a finite-state system that increases a counter i only until a value of 1 - we assume an initial valuation $i = 0$. Third, the other STS example (figure 2c) showing an infinite-state system increasing a counter i infinitely - we again assume an initial valuation of $i = 0$.

Along with these three transition systems, we try to determine whether the following properties are true: with the LTS whether state s_1 or whether state s_2 are reachable; with the first STS, whether location variable i can reach the values 1 or 2; with the second STS, whether location variable i can reach the values 1 or -1. With each LTS or STS, the first property is true while the second one is false.

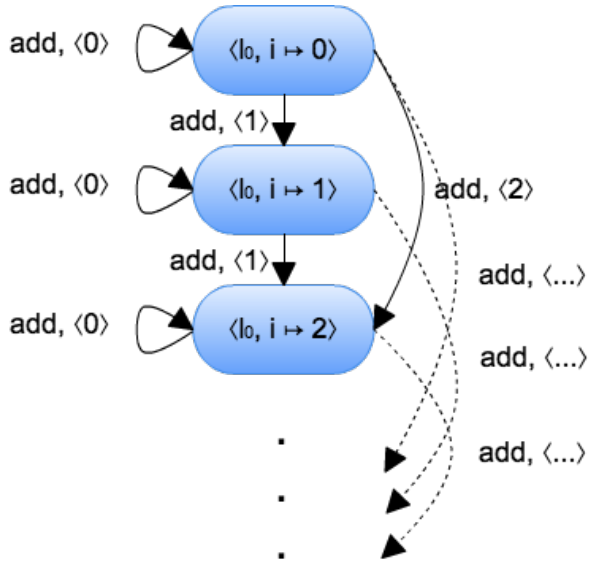


(a) Graphical description

$\mathcal{S}_{calc} = \langle L, l_0, \mathcal{V}, \mathcal{I}, \Lambda, \rightarrow \rangle$ is:

L	$\{l_0\}$
$\mathcal{V}$	$\{int\ i\}$
$\mathcal{I}$	$\{int\ \_x\}$
Λ	$\{add, sub, mult, div\}$
$\rightarrow$	$\{(l_0, add, \_x >= 0, i := i + \_x, l_0),$
	$(l_0, sub, \_x >= 0, i := i - \_x, l_0),$
	$(l_0, mult, true, i := i * \_x, l_0),$
	$(l_0, div, \_x! = 0, i := i/\_x, l_0)\}$

(b) Formal description



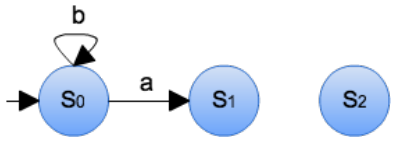
(c) Graphical Semantics

$[\![\mathcal{S}]\!]_\iota = \langle \{\langle l_0, \mathfrak{U}^{\mathcal{V}} \rangle\}, (l_0, i \mapsto 0), \Sigma, \rightarrow \rangle$, where

$$
\begin{aligned}
\Sigma \quad &= \quad \{\ldots, \langle add, \langle -1 \rangle\rangle, \langle add, \langle 0 \rangle\rangle, \langle add, \langle 1 \rangle\rangle, \ldots\} \\
\rightarrow \quad &= \quad \{\langle \langle l_0, i \mapsto 0 \rangle, \langle add, \langle 0 \rangle\rangle, \langle l_0, i \mapsto 0 \rangle\rangle, \\
&\quad\ \langle \langle l_0, i \mapsto 0 \rangle, \langle add, \langle 1 \rangle, \langle l_0, i \mapsto 1 \rangle\rangle, \\
&\quad\ \langle \langle l_0, i \mapsto 0 \rangle, \langle add, \langle 2 \rangle\rangle, \langle l_0, i \mapsto 2 \rangle\rangle, \\
&\quad\ \ldots \\
&\quad\ \langle \langle l_0, i \mapsto 1 \rangle, \langle add, \langle 0 \rangle\rangle, \langle l_0, i \mapsto 1 \rangle\rangle, \\
&\quad\ \langle \langle l_0, i \mapsto 1 \rangle, \langle add, \langle 1 \rangle\rangle, \langle l_0, i \mapsto 2 \rangle\rangle, \\
&\quad\ \langle \langle l_0, i \mapsto 1 \rangle, \langle add, \langle 2 \rangle\rangle, \langle l_0, i \mapsto 3 \rangle\rangle, \\
&\quad\ \ldots \\
&\quad\ \langle \langle l_0, i \mapsto 2 \rangle, \langle add, \langle 0 \rangle\rangle, \langle l_0, i \mapsto 2 \rangle\rangle, \\
&\quad\ \ldots\}
\end{aligned}
$$

(d) Formal Semantics

Figure 1: STS of a calculator



(a) one state unreachable

(b) loop with restriction

(c) loop without restriction

Figure 2: STS examples

□

4. Model Checking of STSs with SMT solvers

We consider three different approaches to Model Checking reachability properties of STSs with SMT solvers. To determine the feasibility of using SMT solvers in the process, we vary the degree of SMT solver involvement in the considered approaches. As a reference point, we consider an explicit state model checker using no SMT solver. At the other extreme, we include a model checker entirely implemented in an SMT solver. Then, we consider a hybrid solution consisting of an external algorithm using an SMT solver only for parts of the Model Checking problem.

First of all, we will present the result to our solution involving no SMT solvers (section 3.1). Second, we show an approach to formulate the Model Checking problem entirely as a query to the SMT solver (section 3.2). Third, we describe a hybrid solution that uses an SMT solver only for part of the problem.

4.1. Explicit State Model Checking

Explicit State Model Checking, the first approach that we present, encodes the whole Model Checking problem in an external algorithm, i.e. it does not involve the use of an SMT solver. Therefore, the control over the search lies entirely on the side of the algorithm. Our version of this approach is based on the Computation Tree Logic (CTL) Model Checking algorithm (see e.g. [BK08] algorithm 13). It takes an LTS and a reachability property as input and tries to verify the reachability property for the given LTS. The search to determine whether there is a valid path from an initial state to the states described by the property is conducted backwards and in a breadth-first manner. This search is unbounded, i.e. there is no maximum path length. For a finite set of states, the search is finite and always terminates for reachability properties. However, the path to a reachable state is not administered. Thus, if a state as described by the property is found to be reachable, no concrete witness path leading to that state is produced.

We first present the algorithm. Second, we describe our implementation of the algorithm. Third, we discuss its termination behaviour.

The algorithm

Our Explicit State Model Checking algorithm *ctl_mc* (listing 1) takes as input an LTS $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$ and a reachability property P and determines whether the states described by property P are reachable from the initial state s_0. To that end, the function *sat_set* calculates the set of all states from which the states described by property P are reachable. This, in turn, involves the calculation of the post-image of states with the function *Post*.

A set theoretic approach is taken to determine whether the states described by property P are reachable. Sets in our case contain states to which the operations subset, equality, element of, union, and intersection are applied.

Listing 1 Explicit State Model Checking

```
 1: ctl_mc(ℒ = ⟨S, s0, Σ, →⟩, Property P)
 2:     return s0 ∈ sat_set(ℒ, P)
 3:
 4: sat_set(ℒ = ⟨S, s0, Σ, →⟩, StateSet T)
 5:     T' = T ∪ {s | s ∈ S ∧ Post(ℒ, s) ∩ T ≠ ∅}
 6:     if T = T'
 7:         return T
 8:     else
 9:         return sat_set(ℒ, T')
10:
11: Post(ℒ = ⟨S, s0, Σ, →⟩, State s)
12:     return {s' | s → s'}
```

Given an LTS $\mathcal{L}$ and a reachability property P, `ctl_mc` determines whether the initial state s_0 is among the states that are backward-reachable from the states described by property P (line 2). This means to check whether there is a path from s_0 to the states described by P. To this end, `sat_set`

(line 4) recursively computes all states backward-reachable from the states described by P. With each recursive step, the set T of states that were already found to be backward-reachable from P is extended with states which have a transition to a state in T (line 5). If there are no new states found having such a transition, all states that are backward-reachable from P have been found (line 6 & 7). Otherwise, the search is continued recursively (line 8 & 9). States that are reached by one transition from a certain state are computed by `Post` (line 11 & 12).

Implementation

We implemented our explicit state model checker in the programming language Haskell. An LTS is represented by the algebraic data type *LTS* consisting of a set of states, an initial state and a set of transitions. A reachability property is represented by the algebraic data type *Property*. The functions of listing 1 are represented with Haskell functions of the same names. Our Haskell implementation is given in a single file of 78 lines, including comments and white space.

Termination behaviour

For a finite set of states S in LTS $\mathcal{L}$, the algorithm always terminates: First of all, the function `Post`$(\mathcal{L}, State\ s)$ computes the set of states reachable from s which is finite because S is finite. Second, `sat_set`$(\mathcal{L}, StateSet\ T)$ recursively extends the set T with states from S until no new states are added, i.e. $T = T'$. Hence, $T \subseteq S$ and as S is finite, this computation always terminates. Third, `ctl_mc` executes the check whether s_0 is an element of `sat_set`. As the latter is finite, this check also always terminates. Considering the termination behaviour of `ctl_mc` on the earlier described LTS (example 2a), we see that for property $P = s_1$, the calculated `sat_set` is $\{s_0, s_1\}$. As the initial state s_0 is among those states, the algorithm yields a positive answer, i.e. state s_1 is reachable. For property $P' = s_2$ however, `sat_set` yields $\{s_2\}$. As the initial state s_0 is not among those states, the reachability of state s_2 is answered negatively.

4.2. SMT-only Model Checking

SMT-only Model Checking, the second approach that we present, encodes the whole Model Checking problem as one SMT solver query, i.e. it does not involve an external algorithm. Therefore, the control over the search lies entirely within the SMT solver. This approach is based on research of Milicevic and Kugler [MK11]. It takes an STS and a reachability property as input and tries to verify the reachability property for the given STS. Reachability properties in this case describe a state by a formula over a location and/or a valuation of location variables. The search to determine the reachability of a state described by the property is done in forward direction, breadth-first, and unbounded, i.e. there is no maximum path length. Furthermore, the search is possibly infinite due to the infinite state space and possible infinite loops of STSs which can cause non-termination. Moreover, if a state as described by the property is found to be reachable, a valid path to that state is reported.

We first describe the algorithm abstractly. Second, we describe its concrete implementation. Third, we analyse its termination behaviour.

The algorithm

The SMT-only approach takes as input an STS $\mathcal{S} = \langle L, l_0, \mathcal{V}, \mathcal{I}, \Lambda, \rightarrow \rangle$, an initial valuation ι, and a reachability property P. Then, it determines whether a state as described by property P is reachable from the initial state $\langle l_0, \iota \rangle$. To that end, a constraint defines requirements for a valid path from the initial state to a state as described by property P. Then, an SMT solver is used to find a path satisfying these requirements.

To determine whether a state as described by property P is reachable, constraints describing valid paths through the STS are built. We determine whether there is a valid path through an STS for a reachability property P by using the logical concept of satisfiability. The search for a valuation satisfying the constraint is implemented by the SMT solver.

For the constraints we consider a path of states $s_0, \ldots, s_k$, where $k \in \mathbb{N}$ does not have an upper limit in the algorithm. The Model Checking problem is then formulated as a conjunction of three sub-constraints: the initial state constraint $I(s_0)$, the transition constraint $S(s_x, s_{x+1})$ with $0 \leq x < k$, and the reachability property constraint $P(s_k)$. The first, $I(s_0)$, is built using the initial state $\langle l_0, \iota \rangle$ and is true iff s_0 is equal to that initial state. The second, $S(s_x, s_{x+1})$, is built using the switch relation $\rightarrow$ and true iff there is a valid transition between the states s_x and s_{x+1}. The third, $P(s_k)$, is built using property P and is true iff s_x fulfills the reachability property which is built as a formula over a valuation of location variables and the location. These sub-constraints are used to search for a path such that $I(s_0) \wedge S(s_0, s_1) \wedge \ldots \wedge S(s_{k-1}, s_k) \wedge P(s_k)$ for $k \in \mathbb{N}$.

We consider three kinds of reachability properties for STSs: location reachability, location variable reachability, and outgoing gate reachability. First, a *location reachability* property specifies whether a state is reachable in which the STS "resides" in the given location. Second, a *location variable* property specifies whether a state is reachable for a given formula in the location variables holds, e.g. whether at least one of some location variables x_0 and x_1 is positive ($x_0 \geq 0 \vee x_1 \geq 0$). Third, an outgoing gate property specifies whether a state is reachable in which an outgoing transition with a given gate exists and a given formula in its interaction variables holds as well. With example 3.6, we could ask whether a negative value can be added to the internal total by specifying a property to search for a state with an outgoing transition with gate "add" such that $\_x < 0$ (which does not exist).

To translate STSs into the initial-, transition-, and property constraint, we use an algebraic data type *State* to represent the state of a transition system. This state consists of the current location, the outgoing gate, and valuations of all location variables. Location variables are represented by accessors of their own name. For example, we write i(s) for the accessor of location variable i applied to $s \in State$, an instance of our algebraic data type. The location and gate are represented by the special accessors *__loc* and *__gate*, respectively. We use a double underscore to avoid ambiguities with interaction variables.

We demonstrate the encoding of an STS into the initial, transition, and property constraint (listing 2) at the example of the calculator from example 3.6. First, the initial constraint $I(s)$, now with an argument s ranging over *State*, sets the initial location and determines the initial valuation of all location variables.

Listing 2 SMT-only Model Checking

```
 1: I(s) :=      __loc(s) = "l0" ∧ i(s) = 0
 2:
 3: S(s, s') :=  __loc(s) = "l0" ∧ __loc(s') = "l0" ∧        // locations
 4:              __gate(s) = "add" ∧                          // gate
 5:              ∃_x : _x ≥ 0 ∧                               // restriction
 6:                    i(s') = i(s) + _x                      // update
 7:              ∨
 8:              __loc(s) = "l0" ∧ __loc(s') = "l0" ∧        // locations
 9:              __gate(s) = "sub" ∧                          // gate
10:              ∃_x : _x ≥ 0 ∧                               // restriction
11:                    i(s') = i(s) − _x                      // update
12:              ∨
13:              ...
14:
15: P(s) :=     i(s) = 3
16: P'(s) :=    __loc(s) = "l0"
17: P''(s) :=   ∃_x : __gate(s) = "add" ∧ _x > 10 ∧
18:                    ((__loc(s) = "l0" ∧ __gate(s) = "add" ∧ _x ≥ 0) ∨ // switches
19:                     (__loc(s) = "l0" ∧ __gate(s) = "sub" ∧ _x ≥ 0) ∨
20:                      ...)
```

Second, the transition constraint $S(s, s')$ describes all ways to change from state s to s'. This constraint is a disjunction of parts where each part encodes one switch of the STS. Such a part consists of the determination of origin and destination location, the restriction, and the update. For the restrictions and updates, all involved interaction variables are existentially quantified, such that an arbitrary value can be chosen for these otherwise free variables. Furthermore, if there were more location variables, the

update part would not only have to determine the new valuation of the location variable that is part of the update in the STS but all other location variables would have also to be assigned their value in the old state because their valuation in the destination state would be undefined otherwise. Note that non-deterministic behaviour caused by two or more switches is encoded directly into the constraints. The SMT solver is then responsible for finding a valuation and involves all possible behaviour in its search. We left out the encoding of the switches of the transitions for multiplying and dividing for space reasons.

Third, the property constraint describes the state we seek to reach. That state can be described by a formula in the location variables, the location, and the gate with its interaction variables. We show three different kind of reachability properties: First $P(s)$, a constraint that checks if there is a state such that location variable i has the value 3; second $P'(s)$, a constraint that checks if there is a state such that the location is "l_0"; and third $P''(s)$, to check whether there is a state with an outgoing transition of the gate "*add*" and whether there is a value $\_x > 10$ for the interaction variable. Note that for that last kind of reachability properties, the origin location, the gate, and the restriction of switches need to be included because no switch constraint limits the valid transitions to a next state – there is no next state as we constraint the last state of the path.

Implementation

To represent the path, we use the theory of lists in SMT solvers. As the length of a list is not predefined, this yields unbounded Model Checking. Furthermore, through the use of SMT solvers' infinite representations of data types, e.g. integers, become possible. As SMT solver we chose Z3, which is developed at Microsoft Research and also used by Milicevic and Kugler in their research. Although the theory works for STSs, we limited our implementation to LTSs for time constraint reasons. That means in our implementation the state consists only of the location, and the transitions involve only gates – location variables, interaction variables, restrictions, and updates are left out. However, SMT solvers are capable of handling constraints involving all components of STSs as described above (see also the section on the hybrid approach).

The implementation of Milicevic and Kugler [MK11], that we reproduce here, consists of two parts: first, an assertion that includes the initial constraint and refers to the transition constraint. Second, an uninterpreted function for the transition constraint. Note that in Z3 constraints are given to the solver as so-called *assertions*.

In the first part (listing 3), it is stated that the search path has to have at least one element, i.e. one state (line 3). Furthermore, the first element of the search path has to fulfill the initial state constraint (line 4). Finally, every two consecutive elements have to fulfill the transition constraint (line 5). The check whether the state that is currently the end of the path is a state as described by property P is in-lined within the *check_tr* function.

Listing 3 Model Checking with SMT lists

```
1: DEF states:StateList
2:     ASSERT
3:         is_cons(states) ∧
4:         I(head(states)) ∧
5:         check_tr(states)
6:     CHECK
```

In the second part (listing 4), the *check_tr* function is defined to return true on a StateList if and only if for every two consecutive list elements, i.e. states, the transition constraint holds. For all StateLists that are given as an argument to *check_tr* the following has to hold: First, if the state list consists of at least two elements (line 3), then for the first two elements the transition constraint has to be fulfilled (line 4). Second, for all further elements of the list (recursion) the whole constraint has to be fulfilled again (line 5). If the in-lined check for the reachability of the state described by P is fulfilled (line 6), the search is terminated by declaring the list to be finished (line 7). The search will stop in the next recursive call as the initial check will fail (line 3). Otherwise, the list is declared to have more elements (line 8), hence the search continues.

Listing 4 Transition constraint and in-lined reachability check

```
1: DEF check_tr: StateList -> Bool
2:     ASSERT FORALL lst:StateList
3:         if (is_cons(lst) ∧ is_cons(tail(lst))) then
4:             S(head(lst), head(tail(lst))) ∧
5:             check_tr(tail(lst)) ∧
6:             if(P(head(tail(lst))))
7:                 then is_nil(tail(tail(lst)))
8:                 else is_cons(tail(tail(lst)))
9:         :PAT {check_tr(lst)}
```

Z3 does not support recursion natively[1]. Therefore, Milicevic and Kugler used pattern instantiation (line 9) of a forall quantification (line 2) to "simulate" recursion. Pattern instantiation "allows instantiating quantified formulas with ground terms that appear in the current search context based on pattern annotations on quantifiers"[2]. In our case, *check_tr* is a term that includes a quantified formula (starting at line 2). This quantified formula is specified to be instantiated only in case the term *check_tr* occurs (specified at line 9). In other words, every occurrence of *check_tr* works as a "call" to the formula defined inside *check_tr*. In our case, the quantified formula defined inside *check_tr* "calls" *check_tr* itself, hence a recursive call is implemented.

Termination behaviour

For an STS S with a finite state space and without loops of switches the SMT-only approach always terminates: the path length will eventually reach a maximum because of the finite, loop-less state space. Then, for any greater path length the transition constraint S (line 4 in listing 4) in the *check_tr* function will not be satisfiable anymore and the search stops. If a state described by property P was reached before that (line 6), the algorithm stops by declaring the list to end (line 7) such that in the next recursive call recursion stops (line 3). If no such state was found, the recursion stops due to the unsatisfiability of a path longer than the maximum length. For an STS with infinite state space or with loops of switches, however, the SMT-only approach does not terminate if a state as described by property P is not reachable: The path can be extended infinitely and the algorithm is not capable to determine whether a state satisfying property P will show up on that path. See the discussion section for a theoretical consideration of the termination behaviour for the STSs of example 3.7.

We were not able to reliably verify properties using our implementation of SMT-only Model Checking. Concerning the LTS in example 3.7 (figure 2a), for the reachability of state s_1 Z3 yielded the result "unknown" on the satisfiability of the constraint. Furthermore, Z3 produced a *candidate model* which was invalid for the constraint. By adding a constraint to define the exact path length, the candidate model became valid. However, the result remained "unknown". Checking a universally quantified formulas that is instantiated on occurrence of a pattern is known[2] to possibly cause the result "unknown". For the reachability of state s_2, Z3 correctly yielded the result "unsat", i.e. the state s_2 is unreachable.

4.3. Hybrid SMT Model Checking

Hybrid SMT Model Checking, the third approach that we present, encodes a part of the Model Checking problem into an external algorithm which then uses an SMT solver to solve the remaining part. Therefore, the control over the search lies partly on the side of the external algorithm and partly on the side of the SMT solver. The version we present is based on the work of Ghilardi et al. [GNRZ08]. It takes an STS and a reachability property as input and tries to verify the reachability property for the given STS. The search to determine whether there is a path from the initial state to a state as described by the property is conducted backwardly and in a breadth-first manner. This search is unbounded, i.e. there is no maximum path length. Furthermore, the searched state space is possibly infinite due to the infinite semantics of STSs. Therefore, this approach does not always terminate for the given reachability property. Moreover, the path to a reachable state is administered such that a witness path can be

[1] http://rise4fun.com/z3/tutorial
[2] http://rise4fun.com/Z3Py/tutorial/advanced

produced. As this hybrid algorithm performs Symbolic SMT STS unbounded Model Checking, we call it S3UMC.

First, we describe the S3UMC algorithm formulated in set theory for illustrative purposes and then translate it into an SMT approach. Furthermore, we describe a simplification of the reachability and fixpoint checks. Then, we describe the implementation of our prototype, and finally, we analyse the termination behaviour.

The hybrid approach in set theory

The hybrid Model Checking algorithm takes as input an STS $\mathcal{S} = \langle L, l_0, \mathcal{V}, \mathcal{I}, \Lambda, \rightarrow \rangle$, an initial valuation ι, and a reachability property P. Like with the SMT-only approach, the Model Checking problem is encoded into constraints defining requirements for paths whose validity an SMT solver can determine. For illustration purposes, we first describe the algorithm using set theory, i.e. constraints are represented by sets instead of logical formulas.

STSs are translated into initial (I), transition (S), and property constraint (P), exactly as described for the SMT-only approach. Then, using these three constraints, a search for the existence of a path $s_0, \ldots, s_n$ is conducted such that $I(s_0) \wedge S(s_0, s_1) \wedge \ldots \wedge S(s_{n-1}, s_n) \wedge P(s_n)$ for $n \in \mathbb{N}$. Unlike with the SMT-only approach, the length n of the path is incremented by an external algorithm. Besides this check for the reachability of a state as described by property P, every time the path length is incremented, an attempt is made to determine whether the state space has been searched exhaustively by employing a *fixpoint* check. If the state space is found to be searched exhaustively, the algorithm stops and reports that a state as described by property P is unreachable.

We again consider three types of reachability properties: location reachability, location variable reachability, and outgoing gate reachability. For details on these properties, see the section on SMT-only Model Checking.

Definition 4.1

Listing 5 S3UMC in set theory

```
 1: function S3UMC(S, I, P)
 2:     int i := 0
 3:     repeat
 4:         if BR^i(S, P) ∩ I(s) ≠ ∅
 5:             return true
 6:         else
 7:             i := i+1
 8:     until BR^i(S, P) = BR^{i-1}(S, P)
 9:
10:     return false
11: end
```

where

1. $\mathrm{BR}^n(S,P) := \bigcup_{j=0}^{n} \mathrm{Pre}^j(S,P)$
2. $\mathrm{PreIm}(S,P) := \{s \mid s \rightarrow s' \wedge P(s')\}$
3. $\mathrm{Pre}^0(S,P) := P$
4. $\mathrm{Pre}^{n+1}(S,P) := \mathrm{PreIm}(S, \mathrm{Pre}^n(S,P))$

□

S3UMC (listing 5) is the core algorithm to check for the reachability of states satisfying property P. S3UMC first executes a *reachability check* to determine if there is a path of length i ending in a state that satisfies property P and starting in a state that satisfies the initial constraint I (line 4). For example, in the first loop iteration this check succeeds if and only if one of the initial states satisfies the property. If the reachability check succeeds, the algorithm stops and reports the satisfiability of property P (line 5). Otherwise, the path length is increased for the next loop iteration (line 7).

Finally, the loop condition (line 8) is a *fixpoint check* to determine whether additional states can be reached by incrementing the path length. This check determines if all states reachable by paths of length $i+1$ are already reachable by paths of length i. If the fixpoint check is true, the loop stops and the algorithm reports the unsatisfiability of property P (line 10). Otherwise, the loop continues.

$\mathrm{BR}^n(S,P)$ symbolically describes all states backward-reachable from states satisfying P by paths of length n or less. $\mathrm{PreIm}(S,P)$ describes the pre-image of P under S, i.e. all states that are backward-reachable in one step from states satisfying property P. $\mathrm{Pre}^0(S,P)$ and $\mathrm{Pre}^{n+1}(S,P)$ recursively describe states backward-reachable in paths of length 0 and $n+1$, respectively.

The hybrid approach in Satisfiability Modulo Theories

For the reachability and fixpoint check, an SMT solver determines their satisfiability. In the following, we describe a simplified version of Ghilardi's original algorithm [GNRZ08]. For this simplified version, we first translated Ghilardi's original algorithm into an abstract version. Then, we simplified this abstract algorithm slightly. For that abstract version and a proof of equivalence of the abstract and the simplified version, see the appendix. In the following, we only show the simplified version.

Definition 4.2

Listing 6 S3UMC with SMT constraints

```
1: function S3UMC(S, I, P)
2:     int i := 0
3:     repeat
4:         if SAT(BR^i(S,P) ∧ I(s))
5:             return true
6:         else
7:             i := i+1
8:     until ¬SAT(¬(BR^i(S,P) ⟺ BR^{i−1}(S,P)))
9:
10:    return false
11: end
```

where

1. $\mathrm{BR}^n(S,P) := \bigvee_{j=0}^{n} \mathrm{Pre}^j(S,P)$
2. $\mathrm{PreIm}(S,P) := \exists s'.S(s,s') \wedge P(s')$
3. $\mathrm{Pre}^0(S,P) := P$
4. $\mathrm{Pre}^{n+1}(S,P) := \mathrm{PreIm}(S, \mathrm{Pre}^n(S,P))$

□

Algorithm S3UMC as in definition 4.2 is identical to that of definition 4.1, except for the substitution of the reachability- and fixpoint checks by constraints using the definitions of $\mathrm{BR}^n(S,P)$ and $\mathrm{PreIm}(S,P)$.

The set theoretic approach relies on sets of states, on the operations 'elements of', intersection, union and equality, and on the special empty set. The constraint logical approach, however, parametrises formulas with one (or more) state(s) such that the formula is satisfiable (SAT) only for certain concrete states. A set S is translated to a constraint $C(s)$ such that $C(s)$ is true if and only if $s \in S$. The operations intersection and union are translated with the logical 'and' and the logical 'or', respectively. Furthermore, the check for the emptiness of a set is replaced by the non-existence of a state satisfying the constraint. Hence, the pre-image PreIm is translated into the search for a state such that the transition constraint holds for that state s and a state s' for which also the property is satisfied. Furthermore, $\mathrm{BR}^n(S,P)$, describing the states backward-reachable in n steps or less, has been translated from a union of sets into a logical disjunction of a translation of those sets.

We describe the sizes of the initial, transition, and property constraint as $|I|$, $|S|$, and $|P|$, respectively. Then, the size of $\mathrm{PreIm}(S,P)$ is $|S|+|P|$ and the size of $\mathrm{Pre}^{n+1}(S,P)$ is $|S| + |\mathrm{Pre}^n(S,P)| = n|S| + |P|$ because $|\mathrm{Pre}^0| = |P|$. Finally, the size of $\mathrm{BR}^n(S,P)$ is $\sum_{j=0}^{n} |\mathrm{Pre}^j(S,P)| = \sum_{j=0}^{n} j|S| + |P| = \frac{1}{2}(n+1)(|P| + n|S| + |P|)$. For complexity analysis, we only consider the term with the highest degree, hence, $|\mathrm{BR}^n(S,P)| \approx \frac{1}{2}n^2|S|$.

Reachability check simplification

The check for the reachability of a state as described by property P (line 4, definition 4.2) given by Ghilardi can be reduced in size because we know that the reachability check failed in all previous loop iterations. After all, if this were not the case, the algorithm would have stopped already (line 4/5). Thus, we can assume $\forall j < i : \neg\,\mathrm{SAT}(\mathrm{BR}^j(S,P) \wedge I(s))$.

Proposition 4.3
Let $i, j \in \mathbb{N}$ and $\mathrm{BR}, \mathrm{PreIm}, \mathrm{Pre}^i, I, S, P$ as in definition 4.2. Then:

$$\begin{gathered}\forall j < i : \neg\,\mathrm{SAT}(\mathrm{BR}^j(S,P) \wedge I(s)) \\ \Longrightarrow \\ \mathrm{SAT}(\mathrm{BR}^i(S,P) \wedge I(s)) \iff \mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I(s))\end{gathered}$$

□

Sketch of the proof
We assume (A1) $\forall j < i : \neg\,\mathrm{SAT}(\mathrm{BR}^j(S,P) \wedge I(s))$. On the one hand, we assume $\mathrm{SAT}(\mathrm{BR}^i(S,P) \wedge I(s))$. Then, say $\neg\,\mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I(s))$. By instantiating (A1) with $j = i-1$ and rewriting it together with $\neg\,\mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I(s))$ in a number of steps using lemma 3.2 yields $\neg\,\mathrm{SAT}(\mathrm{BR}^i(S,P) \wedge I(s))$. This is a contradiction with the assumption $\mathrm{SAT}(\mathrm{BR}^i(S,P) \wedge I(s))$. Hence, $\mathrm{SAT}(\mathrm{BR}^i(S,P) \wedge I(s)) \implies \mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I(s))$. On the other hand, as $\mathrm{BR}^i(S,P)$ follows trivially from $\mathrm{Pre}^i(S,P)$ using definition 4.2.1. Then, by rewriting and using lemma 3.2 $\mathrm{SAT}(\mathrm{BR}^i(S,P) \wedge I(s))$ follows from $\mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I(s))$.

□

In other words, within every loop iteration it suffices to check whether there is a state reachable by paths of length exactly i (Pre^i) instead of by paths of length less than or equal to i (BR^i). For a full proof, see the appendix.

The size of the original reachability check is quadratic in the length of the path: $|\,\mathrm{BR}^i(S,P) \wedge I(s)| \approx \frac{1}{2}i^2|S| + |I|$. Our simplified reachability check's size, however, is linear in the length of the path: $|\,\mathrm{Pre}^i(S,P) \wedge I(s)| = i|S| + |P| + |I|$.

Fixpoint check simplification

Ghilardi et al. noted that their fixpoint check can be simplified [GNRZ08]:

Proposition 4.4
Let $i \in \mathbb{N}$ and $\mathrm{BR}, \mathrm{PreIm}, \mathrm{Pre}^i, S, P$ as in definition 4.2. Then:

$$\begin{gathered}\neg\,\mathrm{SAT}(\neg(\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P))) \\ \Longrightarrow \\ \vDash \mathrm{BR}^i(S,P) \iff \mathrm{BR}^{i-1}(S,P)\end{gathered}$$

□

Sketch of the proof
According to definition 4.2.1 $\mathrm{BR}^{i-1}(S,P) \implies \mathrm{BR}^i(S,P)$ holds because all states reachable by paths of length less or equal to $i-1$ ($\mathrm{BR}^{i-1}(S,P)$) are included in those of length less or equal to i ($\mathrm{BR}^i(S,P)$). Combining this with the assumption $\neg\,\mathrm{SAT}(\neg(\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)))$ and rewriting using the relation between satisfiability and validity (lemma 3.2.4) yields $\vDash \mathrm{BR}^i(S,P) \iff \mathrm{BR}^{i-1}(S,P)$. The latter is equivalent to the constraint shown in definition 4.2 again by lemma 3.2.4. For a detailed proof, see the appendix.

□

The size of the original fixpoint check is quadratic in the length of the path: $|\,\mathrm{BR}^i(S,P) \iff \mathrm{BR}^{i-1}(S,P)| = |\,\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P) \wedge \mathrm{BR}^{i-1}(S,P) \implies \mathrm{BR}^i(S,P)| \approx 2\frac{1}{2}i^2|S| + 2\frac{1}{2}(i-1)^2|S| \approx 2i^2|S|$. Ghilardi's simplified fixpoint check is also quadratic but a factor 2 smaller due to the elimination of one implication.

The fixpoint check can, however, be simplified even further:

Proposition 4.5
Let $i \in \mathbb{N}$ and $\mathrm{BR}, \mathrm{PreIm}, \mathrm{Pre}^i, S, P$ as in definition 4.2. Then:

$$\neg\,\mathrm{SAT}(\neg(\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)))$$
$$\iff$$
$$\neg\,\mathrm{SAT}(\neg(\mathrm{Pre}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)))$$

□

Sketch of the proof
According to definition 4.2.1 $\mathrm{BR}^i(S,P) = \mathrm{Pre}^i \vee \mathrm{BR}^{i-1}(S,P)$. As $\mathrm{BR}^{i-1}(S,P) \iff \mathrm{BR}^{i-1}(S,P)$ is trivially valid, it suffices to determine the validity of $\mathrm{Pre}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)$. The equivalence of the satisfiability of these formulas follows from syntactical the relation between satisfiability and validity described in lemma 3.2. For a detailed proof, see the appendix.
□

Similar as with the simplification of the reachability check, it thus suffices to check whether the states reachable with paths of length exactly i (Pre^i), are already reachable by paths of length less than or equal to $i-1$ (BR^{i-1}).

The size of our simplified fixpoint check is also quadratic in the length of the path but a factor 4 smaller than the original fixpoint check: $|\,\mathrm{Pre}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)| \approx i|S| + \frac{1}{2}i^2|S| \approx \frac{1}{2}i^2|S|$.

Implementation

Our prototype implements the S3UMC algorithm (definition 4.2) for the data types boolean and integer using Z3 as SMT solver. To enable the S3UMC algorithm to send constraints to Z3 for checking their satisfiability, we divided our implementation into a client- and a server part (left and right part in figure 3). On the left side, the client part consists of the model checker *S3UMC* itself and a *Z3Client*. On the right side, the server part contains the *Z3Server* and the SMT solver Z3 itself, which communicate via the Z3Py API. Z3Client and Z3Server, in turn, can communicate via standard method calls that are translated to be sent back and forth via a network interface. Besides these main components, we developed a translator from the internal STS data type used by Axini to the STS data type used by S3UMC to be able to check STSs already defined at Axini.

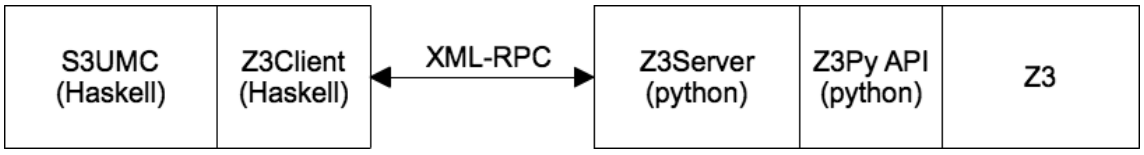


Figure 3: Schematic overview of the prototype implementation

Z3Client and Z3Server expose part of the Z3 interface that is offered by the Z3 API. Exposed functionality includes the definition of custom data types (needed for the State data type, for example), adding and satisfiability-checking of constraints, and requesting a model (valuation) in case of a satisfiable constraint.

As an example, we consider the sending of a constraint C generated by the S3UMC algorithm. First, the S3UMC algorithm hands over constraint C to the Z3Client which offers a method to send constraints

to the SMT solver. Z3Client encodes the constraint into an XML format, yielding C', and sends this encoded constraint C' to Z3Server. Next, Z3Server decodes the constraint C' into a format offered by the Z3Py API. Then it hands the constraint to Z3 via the Z3Py API. Now, S3UMC can either add more constraints or ask Z3 via Z3Client and Z3Server to check the satisfiability of the added constraint(s).

The client side is developed entirely in the programming language Haskell, whereas the server side makes use the programming language python for which Z3 offers the Z3Py API. Communication between the Z3Client and Z3Server is implemented via XML Remote Procedure Calls (XML-RPC).

Termination behaviour

The hybrid approach always terminates for an STS S with a finite state space. Once all states have been reached, the fixpoint check will detect that extending the path length does not reach any new states. Then, the algorithm stops and reports the unreachability of a state as described by property P. If a state as described by property P is reachable within the maximum path length, the reachability check succeeds and the algorithm reports the reachability before terminating due to the fixpoint check.

Concerning the LTS in example 3.7 (figure 2a), our implementation of the hybrid approach is able to verify the reachability of state s_1 and the unreachability of state s_2. Concerning the first STS of example 3.7 (figure 2b), it is also able to verify the reachability of a state with $i = 1$ and the unreachability of a state with $i = 2$. Finally, concerning the second STS of example 3.7 (figure 2b), it is able to verify the reachability of a state with $i = 1$. However, it is not able to verify the unreachability of a state with $i = -1$ because of the unbounded representation of the integer i which is infinitely incremented, starting with $i = 0$.

Validation

We used the calculator STS (example 3.6), an STS calculating a sum of two-folds and three-folds, and an STS modelling a self scan cashier system to validate our implementation of the hybrid approach.

For the calculator STS, we specified two location variable properties involving only the value of an integer. First, we verified that i can assume the values 1 ($P := x = 1$) and -1 ($P' := x = -1$). As expected, both properties were verified, with a witness path of only one step showing that a value of 1 was added or subtracted from the initial value $i := 0$, respectively. Furthermore, we specified two outgoing gate properties. First, we verified that from a state in which the internal total i has the value 1, a value 3 can be added to it. Second, we verified that it is not possible to add the value -3 to the internal total i in a state with $i = 1$. The latter is not possible because the restriction $\_x \geq 0$ on the switch involving the "add" gate prohibits adding negative values.

To demonstrate the power of constraint solving involved in the hybrid Model Checking approach, we describe an STS that calculates a sum of a two-fold and a three-fold (figure 4). This example involves two switches which add a multiple of 2 or 3, respectively, to an internal sum s. By specifying the location variable property $P := s = z$ with $z \in \mathbb{Z}$, we ask the question whether an arbitrary integer z is a sum of a two-fold and a three-fold.

The hybrid approach is able to determine whether an arbitrary integer is the sum of a two-fold and a three-fold for this STS within a path of length three. Using the initial, transition and property constraints, the SMT solver searches for a path such that (a) s has the value 0 in the path's initial state ($s(s_0) = 0$), (b) in the second state a multiple of 2 has been added to s ($s(s_1) = s(s_0) + 2 * x$), (c) in the third state a multiple of 3 has been added to s ($s(s_2) = s(s_1) + 3 * y$)), and (d) s has the value $z \in \mathbb{Z}$ in the third state ($s(s_2) = z$). Combining these constraints yields the equation $z = 2 * x + 3 * y$ which can be solved efficiently by the SMT solver. Furthermore, the question can be generalised to an arbitrary, fixed number n of summands of multiples of $m_0, \ldots, m_n \in \mathbb{Z}$ by creating an STS with a switch and an update mapping for each of the n summands.

To validate the hybrid approach on a realistic model from industry, we used a model of the Scanflow Cash Register Protocol (SCRP) made at Axini. SCRP was developed for a self scan cash register in which the human cashier is substituted by a Scanflow Unit (SFU) that, for example, handles the bar code scanning. Specifically, SCRP describes a communication standard between the Scanflow Unit (SFU) and the Cash Register (CR). For more background information on SCRP see, for example, [dB13]. The SCRP model we used consists of 228 locations, 4 location variables, 541 switches, and 67 different gates.

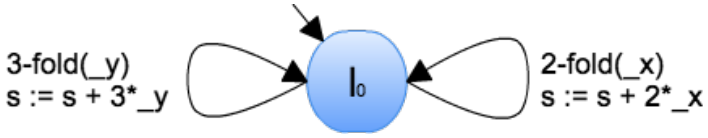


Figure 4: STS to determine two-folds and three-folds

From our brief tests, we conclude that the reachability checks perform reasonably well, while the size of the fixpoint checks grows prohibitively – as expected because of their quadratic complexity.

5. Discussion

5.1. Explicit State Model Checking

Reachability properties are the only properties considered in this research. However, the original CTL algorithm, that our approach is based upon, implements the stronger temporal logic CTL. This temporal logic and its Model Checking algorithm enable the formulation and verification of recursive reachability- and liveness properties (see, for example, [BK08]). Thus, stronger and more expressive properties can be checked using the Explicit State Model Checking approach.

Changing from a breadth-first to a depth-first search is possible. However, the algorithm's structure has to be changed. Let the function *ctl_mc* consider a set of already reached states T. Then, for every state $s \in S \setminus T$ it determines its post-image. If the post-image is found to have common elements with the set of already backward-reachable states T, it is checked whether that state is the initial state. If so, the algorithm terminates reporting the reachability of a state as described by property P. Otherwise, the state s is added to T and a recursive call to *ctl_mc* is started. On returning from a recursive call, the next state of the set $S \setminus T$ is considered. This implements a depth-first search as states that are backward-reachable with one transition from the accumulated states in T are only processed after returning from the recursive calls for all previous states in that set.

With a slight change to the algorithm, it can be changed to a forward search: let the *ctl_mc* function calculate a *sat_set* of all states that are forward-reachable from the initial state s_0. Then, checking whether a state as described by property P is an element of the set returned by *sat_set* determines whether there is a path from the initial state s_0 to a state as described by property P. *sat_set* computes all states forward-reachable from s_0 by accumulating them in a set T, starting with the only element s_0. For each state of the LTS, its pre-image is calculated but the state is only added to set T if and only if its pre-image has common elements with already forward-reachable states in T, i.e. there is a transition from one of the states in T to that state. As in the search with backward direction, *sat_set* terminates if no new states were found to be reachable. Otherwise, a recursive call continues the search for the extended set of forward-reachable states. We see no advantage nor disadvantage of a forward search compared to the backward search.

Changing the input formalism from LTSs to STSs is not possible because the algorithm lacks the capabilities to process switch restrictions, -updates and -gates with interaction variables. However, a translation from STSs to LTSs according to the definition of the semantics of STSs (definition 3.5) is possible. This translation may result in an infinite state space (see example 3.6). To make the state space finite, the universe $\mathfrak{U}$ has to be made finite by choosing finite representations for all data types. Then, the set of states S and the transition relation $\rightarrow$ become finite as well. This finite representation can be used for Explicit State Model Checking of STSs at the cost of pruning the state space by using a finite abstraction of the data types. Therefore, if a reachability property is shown to be false for the finite abstraction of the STS, we cannot conclude that it is also false for the original representation of the STS. After all, a value that has been pruned from the state space may render the states described by the property reachable. Hence, this abstraction may lead to *false negative results* for reachability properties.

The length of the search path can be bounded by introducing a maximum depth for the breadth-first search. For big state spaces this might be suitable to limit the work spent on checking a property. However, a state that is actually reachable by paths longer than the set limit might not be found to be

reachable. Hence, bounding the search path introduces the possibility of reporting *false negatives* for reachability properties.

Not administering paths during the search for reachable states is a disadvantage of the described explicit state approach. This administration can, however, be added by including information on the transitions that lead to a state as described by property P. Let the calculation of the post-image return the complete tuples for transitions (start state, transitions, and goal state). Tuples whose goal state is among the already backward-reachable states could be added directly to the set of already reachable states T. However, a minimisation procedure improves this by combining and filtering tuples to keep only those that describe the shortest path from a certain state to a state as described by the property. Then, the set T consists of tuples of type $S \times [(S \times \Sigma_\tau \times S)]$ describing the backward-reachable state and steps of the path from it to a state described by the property, respectively. Adding the administration of paths enables the reporting of a witness path in case a state described by a reachability property is actually reachable.

Haskell, our choice of programming language, showed to be suitable for representing the needed data types in a concise way. Furthermore, implementing the algorithm was easily achieved using the functional programming paradigm. Nevertheless, a straight-forward translation to other functional programming languages that allow custom declarations of algebraic data types, e.g. F#, is considered possible. Furthermore, a translation to procedural and object-oriented programming languages, e.g. C++ and Java, is perfectly possible as well.

5.2. SMT-only Model Checking

We were not able to reproduce Milicevic and Kugler's approach using the SMT solver Z3 with the SMTLIBv2 interface. Z3 yields the result "unknown" which means that it was unable to derive neither the satisfiability nor the unsatisfiability of the given constraint. Using universally quantified formulas with Z3 can cause the result "unknown" because determining the satisfiability of these formulas is undecidable, i.e. not necessarily terminating for *every* input. Nevertheless, a candidate model is generated which was incorrect in our tests. Strengthening the constraint by defining the exact path length that we searched for yielded correct candidate models, while the derived result remained "unknown". However, limiting the path requires an external algorithm to change this limit in order to achieve unbounded search paths which would no longer yield an SMT-only approach. We conclude that we are doubtful about the reliability of the results and that this approach stretches the capabilities of Z3.

While other SMT-solvers cannot overcome the theoretical undecidability of the satisfiability for constraints involving universal quantification, they may be better suited for solving such constraints. Moreover, a different formulation of the SMT-only approach using specific capabilities of other SMT solvers might lead to better results. Furthermore, options of Z3 to control the quantifier instantiation may also lead to more reliable results[3]. Hence, we conclude that Z3 is not necessarily suitable for the SMT-only approach although further research might improve results with Z3. Despite the theoretical limits, other SMT solvers may be better suited. We did not investigate this any further due to time constraint reasons.

Reproducing the exact result of Milicevic and Kugler using Z3 version 2.0 with the .Net API was possible. For the SimpleWhile example mentioned in their paper, their setup yielded the correct verdict. However, preliminary tests showed that in case a counter example was produced with their setup, the result derived by Z3 was "unknown" as well – including correct candidate models. We did not further investigate the reliability of this result. Hence, we cannot make any statement with respect to the general validity of the candidate models generated by their setup compared to our setup.

Different versions of Z3 on different platforms showed varying results with respect to the derivation of candidate models: Z3 version 4.3.2 (beta) from a nightly build run on Mac OS 10.8.4 did not produce any candidate models; however, Z3 version 4.3.1 compiled and run on Mac OS 10.8.4 did produce candidate models; and Z3 version 4.3.0 32-bit installed with the provided installer on Windows XP produced candidate models as well. Furthermore, changing options and using tweaks forcing Z3 to use different internal solvers can enable it to produce candidate models[4]. We did not investigate this any further.

Our implementation only supports LTSs. However, in the following we give the theoretical termination results for example 3.7 that would have been derivable with an implementation for STSs and correct Z3

[3] http://stackoverflow.com/questions/13198158/proving-inductive-facts-in-z3

[4] http://stackoverflow.com/questions/15806141/keep-getting-unknown-result-with-pattern-usage-in-smtlib-v2-input

derivation of the inductive fact. First, for the LTS (figure 2a), the reachability of state s_1 would have been answered positively, while the reachability of state s_2 would have been answered negatively. Second, for the first STS (figure 2b) – with a finite state space –, the reachability of a state with $i = 1$ would have been answered positively, while the reachability of a state with $i = 2$ would have been answered negatively as well. Third, for the second STS (figure 2c) – with an infinite state space –, the reachability of a state with $i = 1$ would have been answered positively. However, the reachability of a state with $i = -1$ would not have been derivable: the search is started with $i = 0$ and then i is infinitely incremented. Hence, the SMT-only approach will not terminate. Milicevic and Kugler mentioned already that "unfortunately, in some cases our approach cannot prove that no counter example exists (e.g. in the presence of infinite loops in the program)" [MK11].

We only considered reachability properties in this research. However, the SMT-only approach could be extended to recursive reachability properties: consider two properties P_0 and P_1. We search for the reachability of a state as described by P_0 and after that for the reachability of a state as described by P_1. This can be implemented by using the SMT-only approach twice after one another. Compared to CTL Model Checking, the depth of the property recursion has to be known in advance, as it needs to be hard-coded into the implementation due to the lack of external algorithmic control. Considering liveness properties, the SMT-only approach could show the presence of an infinitely long path s by stopping the search on finding a state s_x that has occurred in the path before, i.e. $s_y = s_x$ with $x > y$. After all, the sub-path leading from s_y to s_{x-1} could be repeated infinitely. However, showing the absence of such paths in infinite state spaces requires more sophisticated algorithms.

Changing from a breadth-first to a depth-first search is not possible with this SMT-only approach. We specify requirements for a valid path through the transition system. Then, during the search process the path length is incrementally increased. Thus, all paths of length x are considered before paths of length $> x$ are tried, implementing a breadth-first search. Due to the limited control over the algorithm used by the SMT-solver, trying path length $> x$ before having considered all paths of length $\leq x$ is not possible with the approach we described.

The SMT-only approach can be used to implement a search in backward direction by reversing the direction of the path. A valid path is then specified to start with a state as described by property P, all consecutive pairs of elements of the path have to fulfill the transition constraint S in reversed order, and the last element has to fulfill the initial state constraint I: $P(s_0) \wedge S(s_1, s_0) \wedge \ldots \wedge S(s_k, s_{k-1}) \wedge I(s_k)$ for $k \in \mathbb{N}$. We see no advantage nor disadvantage of changing the direction of the search.

The length of the search path can be bounded for this approach. This were to enforce the termination of the approach for every input at the cost of introducing falsely negative reachability results: states that would be reachable with paths longer than the set boundary are not found to be reachable.

We chose Z3 as SMT solver because it was used by Milicevic and Kugler in their research and because it is actively developed and well-documented. As interface, we chose SMTLIBv2 because it offers a text-file interface, requiring no other programming language compared to other interfaces, like the python or .Net APIs. Specifically, we chose version 2 of the SMTLIB standard because earlier versions did not support the theory of lists yet.

5.3. Hybrid SMT Model Checking

The hybrid Model Checking approach could be extended to recursive reachability properties by chaining the searches for the respective states (see discussion section on SMT-only Model Checking). Considering liveness properties, the hybrid Model Checking approach could also be used to detect the presence of an infinite path (see discussion section on SMT-only Model Checking). However, showing the absence of infinite paths in infinite state spaces, would require the search to stop in infinite state spaces, which is not always the case for the hybrid approach despite a fixpoint check. Hence, we conclude that, like with the SMT-only approach, more sophisticated algorithms are necessary for liveness properties.

Changing from a breadth-first to a depth-first search is not possible with the hybrid approach. As with the SMT-only approach, we define requirements for valid paths through the transition system. By searching for all valid paths of a certain length and incrementally increasing the length of the path, the possibility for a depth-first search is eliminated.

Changing the direction of the search from backward to forward is equally possible like with the SMT-only approach. Exchanging the property- and initial state constraint and furthermore reversing the

transition constraint is sufficient. This yields a search for a path $s_0, \ldots, s_k$ such that $P(s_0) \wedge S(s_1, s_0) \wedge \ldots \wedge S(s_k, s_{k-1}) \wedge I(s_k)$ for $k \in \mathbb{N}$.

The length of the search can be bounded by introducing a maximum path length in the external algorithm. As mentioned earlier, this may lead to falsely reported unreachability of states because states are pruned from the search.

Concerning the termination behaviour, the hybrid approach is not able to verify the unreachability of a state with $i = -1$. Specifically, it does not cover the detection that location variable i will only increase from its initial value 0 and thus never reach the value -1. For this property the algorithm does not terminate because the reachability check will never succeed and the fixpoint check does not cover the detection of infinite paths. In general, for STSs with infinite loops including an update – and therefore an infinite state space – the unreachability of a state that is in fact unreachable cannot be verified.

We chose Z3 because we gained experience with it in the SMT-only approach and because it is well-documented. Haskell was again chosen for its ease to define data types and its concise way to implement algorithms. The choice for Haskell led to the need for developing a way to communicate with Z3. To the best of our knowledge, there is no readily available interface between Haskell and Z3. On the side of Z3, there are three interfaces that we considered: the SMTLIBv2 standard, the .Net API, and the Z3Py API. First, we decided against the text interface of SMTLIBv2 we consider it too slow: For every constraint that needs to be checked, a file has to be created with that constraint and data type declarations. Then, the Z3 executable is called with this file as input. Thus, for every constraint that we want to check the executable has to be started again. Restarting the Z3 executable takes time and therefore we consider this approach as too slow. Second, we decided against the .Net API because it requires to use a proprietary Microsoft development environment that is not available to everyone free of charge. Third, the Z3Py API provided a non-proprietary interface that enabled us to check constraints without the need to start an executable each time. However, we had to develop a custom way to communicate between Haskell and Z3.

Our choice of communication between Haskell and Z3 was an XML-RPC interface. Other options for the communication protocol were JSON-RPC and a socket interface with custom parsers. First, we decided against JSON-RPC because we could find better documentation on XML-RPC for the Haskell side and readily usable server examples for the python side. Second, we decided against a socket interface to exchange string representations of constraints and Z3 commands because of the need to develop a custom parser on either the Haskell or the python side. If we had used the standard serialisation methods for data types in Haskell (read/write functions), a parser on the python side would have been needed to parse the string representation of that serialized data. If we had used serialisation methods on the python side, a parser to translate the string representation back into Haskell data types would have been needed. Furthermore, a framework to call functions for data type declarations, adding and checking of constraints, and requesting of a model from Z3 would have been needed. Moreover, socket programming is labour-intensive and error-prone due to its low-level character. XML-RPC provides the functionality to export functions on the server side which can then remotely be called from the client side as if they were local functions. Moreover, for standard data types, e.g. integers and booleans, data type conversions are also provided. Nevertheless, we had to implement translations for our custom data types for constraints and a way of "bookkeeping" for declared data types and instantiated variables on the Z3 side because variables need to be created before constraints that use them can be sent. Say, for example, that the client side wants to check the satisfiability of a constraint involving the state variable s_0: First, s_0 has to be created on the server (Z3) side. Then, a string representation of the constraint referring to s_0 is sent. During its conversion from the XML-RPC interface to the Z3Py API, this reference to s_0 has to be detected and linked to the actual programming language reference of s_0. Despite the need to manually translate custom data types, we consider XML-RPC the best choice for the communication protocol. Furthermore, every choice involving socket interfaces will need some means of manual translation from Haskell to python data types.

6. Summary

We investigated three Model Checking approaches, varying the degree of SMT solver involvement: the explicit state approach, involving no SMT solving; the SMT-only approach, encoding the whole Model Checking problem in an SMT solver; and the hybrid approach, with an external algorithm that uses an SMT solver for part of the problem.

First of all, more SMT solver involvement yields less control over the search. No SMT solver involvement, however, limits the expressiveness and requires a simplification of the input, hereby rendering native checking of STSs impossible. The combination of both an external algorithm and SMT solvers, the hybrid approach, enables native checking of STS with satisfying control over the search.

We focused on reachability properties for all approaches which they were all able to verify (with problems in case of the SMT-only approach). Checking liveness properties with the described approaches involving SMT solvers is not possible. More sophisticated algorithms are necessary.

Considering the search characteristics, nor the forward nor the backward direction show any clear advantage over the other with either of the approaches. All approaches are capable of an unbounded search and able to administer the search path for witness path production (the explicit state approach only after a change). Also, all described approaches execute the search in a breadth-first manner – only the explicit state approach provides the possibility of a depth-first search.

Considering the approaches separately, the explicit state approach provides a high degree of control over the search but lacks the capability to handle the infinite state space of STSs. The SMT-only approach provides very little control over the search, but it enables checking STSs. With loop-less STSs of finite state space it always terminates, while with infinite state spaces it is unable to verify the unreachability of a given state. In our implementation, the SMT-only approach operates at the limits of SMT solvers and we are doubtful about the reliability of the results. The hybrid approach provides more control over the search and allows checking STSs with their infinite state spaces using the SMT solver within its standard capabilities. Again, with STSs of finite state spaces, it always terminates, while with infinite state spaces it is unable to verify the unreachability of a given state. Compared to previous research, we reduced the size of the involved constraints by one order – quadratic to linear – and a factor of two, respectively. With our implementation we were able to produce reliable results for the data types boolean and integer using the SMT solver Z3.

Because of its high degree of control over the search, its standard-like use of SMT solvers and its capability to fully represent STSs, we conclude that the hybrid approach is best suited for Model Checking STSs using SMT solvers.

7. Conclusion and future work

We conclude that it is possible to build a model checker for reachability properties of STSs with all their components using SMT solvers. A hybrid approach in which an external algorithm directs the search and an SMT solver is used for parts which the external algorithm is unable to solve yielded the best results. Being able to verify properties of STS and thereby validating them is a promising improvement to gaining confidence in model correctness. In the field of Model-Based Testing, translating system requirements into a model is a manual task. Verifying properties of the model and the increased confidence in correctness improve the quality of the testing task. Furthermore, the expressive power of STSs combined with a Model Checking approach that is capable to handle infinite state spaces is also promising for other systems that typically need to be verified, e.g. security and communication protocols.

Considering the research question(s), we are able to show how to translate the whole STS structure into an SMT Model Checking problem. That is, we are able to represent locations and switches including updates, gates, and restrictions with an existential quantification of interaction variables in gates. The translation into an SMT Model Checking problem, however, should not be done in a single constraint. Concerning data types, we are able to handle integers and booleans, although more data types that are supported by SMT solvers could be added. Furthermore, while we exploited a specific SMT solver to the limits of its capabilities, a more standard use yielded more reliable results. Finally, another approach with even less SMT solver involvement needs to be investigated to determine whether representing only combinations of switch restrictions to the SMT solver yields better results than having it search for a

path as well – we suggest such a research project below.

We could extend our work on the SMT-only and the hybrid approach along various lines: On a conceptual level, we could extend the considered properties, e.g. by investigating which parts of a chosen temporal logic can be used in Model Checking with SMT solvers. Furthermore, we could investigate the termination behaviour, especially that of the hybrid approach, more closely to improve its behaviour for infinite state spaces. Moreover, validating our research in a big and realistic case study could provide more insights into weaknesses and pointers for improvements and extension. On an implementation level, we could repeat the research with different SMT solvers for comparison, investigate the use of options that Z3 provides, and try other ways of communicating with Z3. Furthermore, we could improve the performance of the communication with Z3.

As another interesting change, we could replace the SMT solver by an automated theorem prover. We are very curious how the change from the notion of satisfiability with SMT solver to the notion of validity with theorem provers would improve or worsen the expressiveness and capabilities of Model Checking STSs. Especially, the possibilities to symbolically reason about constraints might overcome the shortcomings in termination behaviour of the described approaches.

As one concrete path to follow, we suggest to investigate other hybrid approaches with even less SMT solver involvement than in the presented hybrid approach. Such approaches should check reachability properties (or even liveness properties) of STSs by executing an unbounded search in infinite state spaces and determine witness paths in case of the reachability of states. Extending the influence of the external algorithm increases the control over the search, enabling, for example, a depth-first search. Hence, a comparison between a depth-first and a breadth-first search becomes possible. Furthermore, the external algorithm could make better use of the structure of an STS: for example, by specifying only a subset of all switches or a subset of all locations in a constraint that is given to the SMT solver. In other words, the external algorithm could guide the creation of possible paths through the STS and use the SMT solver only for solving constraints that arise through formulas in switch restrictions and updates. Specifying smaller parts of an STS in constraints also reduces their size and thus can improve the applicability to large STSs as the SCRP model described above. We are curious on how the increased control over the search influences the capability to check other types of properties and the termination behaviour in infinite state spaces.

A. Proof Preliminaries

Proof (lemma 3.2)

1. Assume $\mathrm{SAT}(\phi \wedge \psi)$. Then, there is a model M such that $M \vDash \phi \wedge \psi$. Take the model M' equal to M except that all variables which are not in the free variables of ψ are removed. Then, for this model still $M' \vDash \phi$. Hence, $\mathrm{SAT}(\phi)$. Analogously for $\mathrm{SAT}(\psi)$. Hence, $\mathrm{SAT}(A) \wedge \mathrm{SAT}(\psi)$.
2. Assume $\mathrm{SAT}(\phi \vee \psi)$. Then, there is a model M such that $M \vDash \phi \vee \psi$. Thus, under this model either ϕ or ψ (or both) evaluates to true. Thus, $\mathrm{SAT}(\phi)$ or $\mathrm{SAT}(\psi)$, according to definition 3.2. Hence, $\mathrm{SAT}(\phi \vee \psi) \implies \mathrm{SAT}(\phi) \vee \mathrm{SAT}(\psi)$.
   Now assume $\mathrm{SAT}(\phi)$. Then, there is a model M such that $M \vDash \phi$. Take the model M' equal to M, except that M' is extended with all free variables of ψ that do not occur in ϕ and M' assigns random values to those 'new' free variables. Then, $M' \vDash \phi$ and, vacously, $M' \vDash \phi \vee \psi$. Thus, $\mathrm{SAT}(\phi \vee \psi)$, according to definition 3.2. Analogously for $\mathrm{SAT}(\psi)$. Hence, $\mathrm{SAT}(\phi) \vee \mathrm{SAT}(\psi) \implies \mathrm{SAT}(\phi \vee \psi)$.
3. Assume $\mathrm{SAT}(\phi \implies \psi)$. Rewriting yields the equivalent $\mathrm{SAT}(\neg\phi \vee \psi)$. Then, using 3.2.2, this is equivalent to $\mathrm{SAT}(\neg\phi) \vee \mathrm{SAT}(\psi)$.
4. Assume $\neg\,\mathrm{SAT}(\neg\phi)$. Now say $\nvDash \phi$, i.e. there is no model such that ϕ evaluates to true. But then, under every model M $\neg\phi$ evaluates to true. Thus, $M \vDash \neg\phi$. This yields $\mathrm{SAT}(\neg\phi)$ according to definition 3.2. However, this contradicts our first assumption. Thus, $\vDash \phi$.
   Assume $\vDash \phi$. Now say $\mathrm{SAT}(\neg\phi)$, thus there is an M such that $M \vDash \neg\phi$. However, due to our first assumption, for the same M it holds that $M \vDash \phi$. Contradiction. Hence, $\neg\,\mathrm{SAT}(\neg\phi)$.

□

B. Adaptation of Ghilardi's algorithm

The original algorithm BReach of Ghilardi differs from the final algorithm S3UMC (definition 4.2) in several ways: First of all, we slightly simplified its structure. Second, we simplified the reachability check. Third, we simplified the fixpoint check. In the following, we give proofs for these three simplifications.

B.1. Simplification of Ghilardi's algorithm

The original algorithm BReach of Ghilardi differs from the simplified algorithm S3UMC (definition 4.2) in its structure: S3UMC executes all reachability checks inside the loop, while BReach executes the first reachability check outside of the loop.

Here, we first describe an abstract version of Ghilardi's original algorithm BReach [GNRZ08]. Then, we show that our simplified algorithm is equivalent to that abstract version of the original algorithm.

Definition B.1

```
 1: function BReach(S,I,P)
 2:     if SAT(BR^0(S,P) ∧ I(s))
 3:         return true
 4:
 5:     int i := 0
 6:     repeat
 7:         if SAT(BR^{i+1}(S,P) ∧ I(s))
 8:             return true
 9:         else
10:             i := i+1
11:     until ¬SAT(¬(BR^{i+1}(S,P) ⟹ BR^i(S,P)))
12:
13:     return false
14: end
```

where

1. $\mathrm{BR}^n(S,P) := \bigvee_{j=0}^{n} \mathrm{Pre}^j(S,P)$
2. $\mathrm{PreIm}(S,P) := \exists l'.S(s,s') \wedge P(s')$
3. $\mathrm{Pre}^0(S,P) := P$
4. $\mathrm{Pre}^{n+1}(S,P) := \mathrm{PreIm}(S, \mathrm{Pre}^n(S,P))$

□

Proposition B.2
The algorithms BReach (definition B.1) and S3UMC (definition 4.2) are equivalent, i.e. they terminate on the same input, and if they terminate, they yield the same output when given the same input.

□

Proof (proposition B.2)
First, we show that the fixpoint check in the loop conditions of both algorithms are equivalent. Second, we show that both algorithms also execute the same reachability checks. From these two facts we deduce that the algorithm are equivalent as described in the proposition.

The fixpoint checks of BReach executed in the loop condition of iteration $i = 0, 1, 2, \ldots$ are $[\mathrm{BR}^2(S,P) \iff \mathrm{BR}^1(S,P),\ \mathrm{BR}^3(S,P) \iff \mathrm{BR}^2(S,P),\ \ldots]$. For S3UMC, however, these are $[\mathrm{BR}^1(S,P) \iff \mathrm{BR}^0(S,P),\ \mathrm{BR}^2(S,P) \iff \mathrm{BR}^1(S,P),\ \mathrm{BR}^3(S,P) \iff \mathrm{BR}^2(S,P),\ \ldots]$. That is, BReach leaves out the fixpoint check for the equivalence of states reachable by paths of length 1 and 0. However, in case that states reachable by paths of length 1 are equivalent to those of length 0, BReach does not find any path in iteration $i = 1$ that it had not found in iteration $i = 0$ already. Furthermore, it terminates in iteration $i = 1$ because the fixpoint check in this case is in fact a repeated application of the function on its smallest fixpoint and, therefore, yields still the fixpoint. Hence, BReach might need one iteration more than S3UMC to terminate. Other than that, both algorithms execute the same fixpoint checks.

The executed reachability checks of both algorithms are $SAT(\mathrm{BR}^0(S,P) \wedge I(s)$, $\mathrm{SAT}(\mathrm{BR}^1(S,P) \wedge I(s)$, $\mathrm{SAT}(\mathrm{BR}^2(S,P) \wedge I(s)$, ... The only difference is that BReach executes the reachability check for states reachable by paths of length 0 outside of the loop (line 2, definition B.1), while S3UMC does so inside the loop (line 4, definition 4.2). □

B.2. Simplification of Ghilardi's reachability check

Proof (proposition 4.3)
Assume (A1) $\forall j < i : \neg\,\mathrm{SAT}(\mathrm{BR}^j(S,P) \wedge I)$.

Assume $\mathrm{SAT}(\mathrm{BR}^i(S,P) \wedge I)$. Now say, $\neg\,\mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I)$. Instantiating (A1) with $j = i-1$ yields $\neg\,\mathrm{SAT}(\mathrm{BR}^{i-1}(S,P) \wedge I)$. Then

$$
\begin{array}{lll}
 & \neg\,\mathrm{SAT}(\mathrm{BR}^{i-1}(S,P) \wedge I) \wedge \neg\,\mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I) & \\
\iff & \neg\,\mathrm{SAT}(\bigvee_{j=0}^{i-1} \mathrm{Pre}^j(S,P) \wedge I) \wedge \neg\,\mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I) & \text{(def 4.2.1)} \\
\iff & \neg(\mathrm{SAT}(\bigvee_{j=0}^{i-1} \mathrm{Pre}^j(S,P) \wedge I) \vee \mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I)) & \text{(de Morgan)} \\
\iff & \neg\,\mathrm{SAT}((\bigvee_{j=0}^{i-1} \mathrm{Pre}^j(S,P) \wedge I) \vee (\mathrm{Pre}^i(S,P) \wedge I)) & \text{(lemma 3.2.2)} \\
\iff & \neg\,\mathrm{SAT}((\bigvee_{j=0}^{i-1} \mathrm{Pre}^j(S,P) \vee \mathrm{Pre}^i(S,P)) \wedge I) & \text{(distributivity of } \wedge \text{ over } \vee) \\
\iff & \neg\,\mathrm{SAT}(\bigvee_{j=0}^{i} \mathrm{Pre}^j(S,P) \wedge I) & \\
\iff & \neg\,\mathrm{SAT}(\mathrm{BR}^i(S,P) \wedge I) & \text{(def 4.2.1)}
\end{array}
$$

which contradicts our assumption $\mathrm{SAT}(\mathrm{BR}^i(S,P) \wedge I)$. Hence, $\mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I)$.

Now, assume $\mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I)$.

$$
\begin{array}{lll}
 & \mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I) & \\
\implies & \mathrm{SAT}(\bigvee_{j=0}^{i-1} \mathrm{Pre}^j(S,P) \wedge I) \vee \mathrm{SAT}(\mathrm{Pre}^i(S,P) \wedge I) & (\vee \text{ introduction}) \\
\iff & \mathrm{SAT}((\bigvee_{j=0}^{i-1} \mathrm{Pre}^j(S,P) \wedge I)) \vee (\mathrm{Pre}^i(S,P) \wedge I)) & \text{(lemma 3.2.2)} \\
\iff & \mathrm{SAT}((\bigvee_{j=0}^{i-1} \mathrm{Pre}^j(S,P) \vee \mathrm{Pre}^i(S,P)) \wedge I) & \text{(distributivity of } \wedge \text{ over } \vee) \\
\iff & \mathrm{SAT}(\bigvee_{j=0}^{i} \mathrm{Pre}^j(S,P) \wedge I) & \\
\iff & \mathrm{SAT}(\mathrm{BR}^i(S,P) \wedge I) & \text{(def 4.2.1)}
\end{array}
$$

□

B.3. Simplification of Ghilardi's fixpoint check

Proof (proposition 4.4)
Note that $\mathrm{BR}^{i-1}(S,P) \implies \mathrm{BR}^i(S,P)$ is valid by definition 4.2.1. Hence, by lemma 3.2.4 its negation is unsatisfiable: $\neg\,\mathrm{SAT}(\neg(\mathrm{BR}^{i-1}(S,P) \implies \mathrm{BR}^i(S,P)))$.

Then the following holds:

$$
\begin{array}{lll}
 & \neg\,\mathrm{SAT}(\neg(\mathrm{BR}^{i-1}(S,P) \implies \mathrm{BR}^i(S,P))) \wedge \neg\,\mathrm{SAT}(\neg(\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P))) & \\
\iff & \neg(\mathrm{SAT}(\neg(\mathrm{BR}^{i-1}(S,P) \implies \mathrm{BR}^i(S,P))) \vee \mathrm{SAT}(\neg(\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)))) & \text{(de Morgan)} \\
\iff & \neg(\mathrm{SAT}(\neg(\mathrm{BR}^{i-1}(S,P) \implies \mathrm{BR}^i(S,P)) \vee \neg(\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)))) & \text{(lemma 3.2.2)} \\
\iff & \neg(\mathrm{SAT}(\neg((\mathrm{BR}^{i-1}(S,P) \implies \mathrm{BR}^i(S,P)) \wedge (\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)))) & \text{(de Morgan)} \\
\iff & \neg(\mathrm{SAT}(\neg(\mathrm{BR}^i(S,P) \iff \mathrm{BR}^{i-1}(S,P)))) & \\
\implies & \vDash \mathrm{BR}^i(S,P) \iff \mathrm{BR}^{i-1}(S,P) & \text{(lemma 3.2.4)}
\end{array}
$$

□

The fixpoint check can be simplified even further, as stated in proposition 4.5:

Proof (proposition 4.5)
Using natural deduction, we prove that

$$
\begin{array}{c}
\vDash \mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P) \\
\iff \\
\vDash \mathrm{Pre}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)
\end{array}
$$

Then, the proposition holds by lemma 3.2.4 and the fact that for formulas A, B it holds that $(A \iff B) \iff (\neg A \iff \neg B)$.

Assume (A1) $\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)$. Furthermore, assume $\mathrm{Pre}^i(S,P)$. Then also $\mathrm{Pre}^0(S,P) \vee \ldots \vee \mathrm{Pre}^i(S,P)$ which is equivalent to $\mathrm{BR}^i(S,P)$ by definition 4.2. Thus, $\mathrm{BR}^{i-1}(S,P)$ by using (A1). Thus, $(\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)) \implies (\mathrm{Pre}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P))$.

Assume (A2) $\mathrm{Pre}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)$. Furthermore, assume $\mathrm{BR}^i(S,P)$, which is equivalent to $\mathrm{Pre}^0(S,P) \vee \ldots \vee \mathrm{Pre}^i(S,P)$ by definition 4.2.1. We distinguish these i cases. For case $0 \leq x < i$ we trivially have $\mathrm{Pre}^0(S,P) \vee \ldots \vee \mathrm{Pre}^x(S,P) \vee \ldots \vee \mathrm{Pre}^{i-1}(S,P)$ which is equivalent to $\mathrm{BR}^{i-1}(S,P)$. For case i we have $\mathrm{Pre}^i(S,P)$ and thus by (A2) also $\mathrm{BR}^{i-1}(S,P)$. Thus, for all cases $\mathrm{BR}^{i-1}(S,P)$. Thus, $(\mathrm{Pre}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P)) \implies (\mathrm{BR}^i(S,P) \implies \mathrm{BR}^{i-1}(S,P))$.

□

References

[ABG+12] F Alberti, R Bruttomesso, S Ghilardi, S Ranise, and N Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *CAV'12: Proceedings of the 24th international conference on Computer Aided Verification*. Springer-Verlag, July 2012.

[AMP06] A Armando, J Mantovani, and L Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Model Checking Software*, pages 146–162, 2006.

[Arm09] A Armando. Building SMT-based software model checkers: an experience report. *Frontiers of Combining Systems*, pages 1–17, 2009.

[BCC+99] A Biere, A Cimatti, E M Clarke, M Fujita, and Y Design Automation Conference 1999 Proceedings 36th Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference, 1999. Proceedings. 36th*, March 1999.

[BCD+11] C Barrett, C L Conway, M Deters, L Hadarean, D Jovanović, T King, A Reynolds, and C Tinelli. CVC4. In *CAV'11: Proceedings of the 23rd international conference on Computer aided verification*. Springer-Verlag, July 2011.

[BK08] C Baier and J P Katoen. *Principles of model checking*, volume 26202649. MIT press, 2008.

[BvdPW10] S Blom, J van de Pol, and M Weber. LTSMIN: distributed and symbolic reachability. In *CAV'10: Proceedings of the 22nd international conference on Computer Aided Verification*. Springer-Verlag, July 2010.

[CFMS12] L Cordeiro, B Fischer, and J Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Transactions on Software Engineering*, 38(4):957–974, June 2012.

[CGK+12] S Conchon, A Goel, S Krstić, A Mebsout, and F Zaïdi. Cubicle: a parallel SMT-based model checker for parameterized systems: tool paper. In *CAV'12: Proceedings of the 24th international conference on Computer Aided Verification*. Springer-Verlag, July 2012.

[Cla08] E Clarke. The birth of model checking. *25 Years of Model Checking*, pages 1–26, 2008.

[dB13] Vincent de Bruijn. Model-Based Testing with Graph Grammars. 2013.

[DDM06] B Dutertre and L De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.

[DMB08] L De Moura and N Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[DMB11] L De Moura and N Bjørner. Satisfiability modulo theories. *Communications of the ACM*, 54(9):69, September 2011.

[EC82] E A Emerson and E M Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer programming*, 2(3):241–266, 1982.

[Eme08] E Emerson. The beginning of model checking: A personal perspective. *25 Years of Model Checking*, pages 27–45, 2008.

[FTW06] L Frantzen, J Tretmans, and T Willemse. A symbolic framework for model-based testing. *Formal approaches to software testing and runtime verification*, pages 40–54, 2006.

[GNRZ08] S Ghilardi, E Nicolini, S Ranise, and D Zucchelli. Towards SMT model checking of array-based systems. *Automated Reasoning*, pages 67–82, 2008.

[GR09] S Ghilardi and S Ranise. Model Checking Modulo Theories at work: the integration of Yices with mcmt. *Proc. of AFM*, 9, 2009.

[GRV09] S Ghilardi, S Ranise, and T Valsecchi. Light-weight SMT-based model checking. *Electronic Notes in Theoretical Computer Science*, 250(2):85–102, 2009.

[Hol97] G J Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[Lam77] L Lamport. Proving the Correctness of Multiprocess Programs. *Software Engineering, IEEE Transactions on*, SE-3(2):125–143, 1977.

[McM92] K L McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Carnegie-Mellon University Pittsburgh PA Dept of Computer Science, 1992.

[MK11] A Milicevic and H Kugler. Model checking using SMT and theory of lists. *NASA Formal Methods*, pages 282–297, 2011.

[Mye04] G J Myers. *The Art of Software Testing, Second Edition*. Wiley, 2 edition, June 2004.

[NSVT07] A C D Neto, R Subramanyan, M Vieira, and G H Travassos. A survey on model-based testing approaches: a systematic review. In *WEASELTech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. ACM Request Permissions, November 2007.

[QS82] J Queille and J Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[Rau06] W Rautenberg. *A Concise Introduction to Mathematical Logic*. Springer, September 2006.

[UPL11] M Utting, A Pretschner, and B Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, April 2011.