Slice or Unfold – Experiments on Checking Synchronous Models with Backwards Slicing

Tim Gonschorek * Frank Ortmeier *

* Chair of Software Engineering, Faculty of Computer Science, Otto-von-Guericke-University Magdeburg, Universitätsplatz 2, 39104 Magdeburg (e-mail: tim.gonschorek, frank.ortmeier@ovgu.de).

Abstract: Nowadays, model checking approaches have one major drawback: They suffer from the state space explosion problem. We think one major problem is that the verification tools always must hold some (symbolic) representation, e.g. a BDD or a adjacency list, in the memory during the verification process. In this paper we propose a verification approach that does not need to hold a complete representation while checking the model. This is done by slicing backward the transition rules, which defines the complete state system. Here, we present the basic backward slicing approach for verifying CTL safety properties traversing the state machine by using a given transition model. In the following, we give a little evaluation with special test models and compare the results to the state of the art model checker nuXmv.

Keywords: Monitoring and diagnosis of discrete event systems

1. INTRODUCTION

Software controls more and more embedded systems. Many of these systems control and/or monitor critical hardware. As a consequence, failure of such systems must be avoided. To ensure this, different measures have to be taken. These includes a priori hazard and safety analysis, usage of highly reliable components with redundancies as backups as well as highest quality measures for correctness of software controllers. The most rigorous measure of assuring correctness of some software artifact is by formal analysis.

These formal methods for software analysis can be split into two main categories: interactive verification on the one hand and automatic model checking on the other. While the first is capable of infinite space systems, it requires a very high level of expertise. In contrast to that, model checking is a fully automatic technique (Clarke et al. (1999b)). The main problem of all methods for model checking is the handling of the large state space or its symbolic representation, because they need to have some kind of complete system representation (for example some binary decision diagram) within the memory and execute some model checking algorithm on these.

To overcome these difficulties, we propose an algorithm, which not necessarily needs to compute an abstraction of all states or all reachable states before executing any model checking algorithm. In this paper, we restrict the algorithm to safety critical properties, i.e., the question of (not) reaching a potentially hazardous state. The core idea is to do a guided search through the state space by computing a path from a state violating the specification to an initial state, i.e., to proof that a possible hazardous state is reachable. This is done by using encodings known from SAT based bounded model checking together with ideas taken from program slicing. As a consequence, only the next states are chosen that seem to lead (backwards) to the interesting state, e.g. the initial state. This kind of exploring, where only states are considered that correspond to a certain condition, is called Model Based Slicing (Androutsopoulos et al. (2013)).

For the presented approach, some transition rule based system description language is used as input to explore the state space of interest. In this paper, we take the automata description language developed by Güdemann et al. (2012), SAML (Safety Analysis and Modeling Language).

The remainder of the paper is structured as follows: Section 2 shows a brief overview of the formal notations as well as some running example. Section 3 gives an overview over some related work of the approach. In the following (Sect. 4), we describe the core algorithms. A small numerical evaluation is then presented in Sect. 5 before the paper is concluded and an outlook is given (Sect. 6).

2. BACKGROUND

Before we describe the proposed Backward Model Checking with Slicing algorithm for safety properties, some formal background is given in the following section. It gives a rough introduction to the transition rule based modeling language SAML. Further, we declare what is meant by safety properties and slicing.

2.1 SAML-A Transition Rule Based Modeling Language

The System Analysing Modeling Language (SAML) is a language for conveniently modeling finite state systems
as parallel automata, which has been developed around 2011 and was presented first by Güdemann and Ortmeier (2011). The main goal of System Analysing Modeling Language (SAML) is to allow for conveniently building formal models in a tool-independent manner. Therefore, it combines syntactical elements from qualitative and quantitative model checking languages. Automatic model-transformations then allow for checking a SAML model with various tools, e.g. NuSMV \(^1\), PRISM \(^2\) or UPPAAL \(^3\). Further it provides failure analyzing techniques as the computation of minimal cut sets, presented by Güdemann et al. (2008) or the use of failure sensitive specifications (Ortmeier and Reif (2004)). In this paper, we will use the language SAML for explaining the basic idea of the Backward Model Checking with Slicing (BWMC) algorithm. The model transformations available for SAML then allow to check the same model with different tools. In this paper, we use muXmv, the successor of NuSMV with the same input language, as well as the here proposed algorithm and compare the results. To make the paper self contained, we touch the language only on top level by giving an example program in Listing 1. For further reading on SAML, its syntax and applications we refer to Lipaczewski et al. (2012) and Güdemann and Ortmeier (2010).

In the purpose of this paper, we will use a small running example for presenting basic SAML elements and later for explaining the necessary, formal concepts of the backward slicing approach. This example presents a simple model of a water tank. It consists of the water tank itself as well as a faucet and an outflow. The maximum volume of the tank is 50 liters. Both, the faucet and the outflow have a water flow of one liter per step. Additionally, a user controls the faucet and the outflow valve. Once the water level is 40 liters or higher, it closes the faucet and opens the outflow. If the water level is ten liters or less, the user opens the faucet and closes the outflow.

Let’s have a look at the SAML program of the water tank example. This example would consist of four components (user, tank, faucet, valve). The components user and tank are shown in Listing 1.

In general, a component (user) consists of state variables (faucet_ctrl and valve_ctrl) and transition rules. Their state values are derived from the enumeration openState with the state open and closed. Further, the component have transition rules, covering the transition between states in different situations, e.g. if the water level of the tank (tank.level) is less than 10 liters, the faucet is opened and the outflow is closed (see Listing 1 l.7-l.11). Moreover, they can model some kind of nondeterministic behavior. This is done with the choice keyword. It marks that more then one specific state transition is possible. In the mentioned transition either the faucet is opened and the outflow is closed or the values of the state variables do not change. At each automaton step, one specific transition rule per component is executed synchronous, i.e., all value changes of the state variables happen at the same time.

### Formal Definition

Basically, a SAML model \(M\) is a

\[
SM = (V,D,I,T)
\]

where \(V\) is the set of state variables \(v \in V\), \(D\) is the domain or universe of the interpretation of \(V\), \(I\) is the set of initial values of the state variables \(V\) and \(T\) is the set of transition rules \(t \in T\). Furthermore, each transition rule \(t = (C,A), t \in T\) is a 2-tuple of some formula \(C\) and an assignment \(A\). Additionally, state variables and transition rules are encapsulated in different components, i.e., state variables and transition rules, defining their possible transitions, are within on component.

For any SAML model an equivalent Kripke structure may be automatically derived (by building the product automaton, computing the transition relation from \(T\) and defining the corresponding labeling function). Therefore, a state \(s_i\) is defined as valuation of each state variable \(v_i \in V\) with a value from the corresponding domain \(d_i \in D\) and an initial state value \(i_i \in I\). The product automaton of one transition rule per component, thus, represents a set of transitions, i.e. \(s_i \rightarrow s_j\) with \(s_i, s_j \in S\), \(s_i \models C\) and \(s_j = A(s_i)\). This defines the semantics of the model. However, in contrast to “standard” model checking, we do not want to operate on a representation of the flattened model but rather on the automata model and do slicing on this representation (without doing any flattening of the system).

#### 2.2 Verifying Safety Properties in Computation Tree Logic

Our approach aims to verify given CTL (Clarke et al. (1986)) safety properties. In the CTL context safety properties are specifications, which, informal spoken, state that ”something bad never happens”. As mentioned by Biere et al. (1999b), for CTL this is a specification of the form \(AG\phi\). This specifies that there exists a state \(s_i \in S\), such that on all possible paths \(\pi\) of the Kripke structure \(M = (S,I,R,L)\) \(\phi\) holds globally, i.e. \(s_i, M \models AG\phi\).

Following Katoen and Baier (2008) and Clarke et al. (1999b), to verify such a property formula \(f\) (e.g. \(AG\)), a model checker does not have to show that \(M \models f\) holds

---

**Listing 1. Water tank example as SAML model.**

1 enum openState: [open, closed];
2 component user
3 //state variables
4 faucet_ctrl: openState init openState.open;
5 valve_ctrl: openState init openState.closed;
6 //transition rule
7 tank.level <= 10
8 -> choice: faucet_ctrl' = open
9 & valve_ctrl' = closed
10 + choice: faucet_ctrl' = faucet_ctrl
11 & valve_ctrl' = valve_ctrl;
12 ...
13 endcomponent
14 ...
15 component tank
16 level: [0..50] init 1;
17 faucet.state = open & valve.state = closed
18 & level < 50 -> level' = level + 1;
19 ...
20 endcomponent

---

1 http://nusmv.fbk.eu/
2 http://www.prismmodelchecker.org/
3 http://www.uppaal.org/
in all possible states, but to find a path $\pi$ as witness for $M, \pi \not\models f$, which is equivalent to $M, \pi \not\models \neg f$. This proofs that there exists a trace, reachable from the initial state, where $f$ does not hold. In the context of safety properties, that means that to proof whether an $AG\phi$ holds, we must show that $M \not\models \neg AG\phi$. Taking the relations for CTL operators from Biere et al. (1999b), we have to proof that $M \not\models \neg AF\phi$, i.e., we find no path on which at least $\phi$ holds once.

### 2.3 Transition Unrolling with SMT

To determine transition rules, which are applicable at a current system state as well as for calculating resulting next states, a Satisfiability Modulo Theories (SMT)-solver is used. Simplified spoken, SMT extends the theory of SAT by the use of more complex theories like linear integer or real arithmetic (De Moura and Bjørner (2011)). Therefore transition rules must be transformed into a semantically equivalent SMT program. The basic idea of this transformation had been presented first by Biere et al. (1999a) in the context of bounded symbolic model checking of state machines with SAT-solvers instead of BDDs. However, we use a SAT-solver instead of a SAT-solver, because with SMT it is possible to deal with more complex background theories, i.e., to use quantifier free formulas instead of propositional formulas. The basic extension of using SMT instead of SAT is given by Armando et al. (2006). Since we only touch the idea of transition unrolling, we refer to Biere et al. (2009) for a deeper introduction into SAT and SMT.

In general, the idea of transition unrolling is to transform a given translation into a SMT term from which it can be calculated if the corresponding transition rule is applicable onto the current state and if it does, what is the corresponding next state. In the resulting formula, the next state variable SMT-variables get either an index or the primed form of the variable is used. So, if we take a transition rule of the form $t=(C.A)$ (sec. 2.1.1) we can define the corresponding SMT program as

$$t = C(s) \land A(s')$$

Moreover, if the conjunction with the current model state $s \land t$ is true, the transition rule is applicable for $s$ and the variable assignment of the solver represents the next state $s'$.

### 3. RELATED WORK

First attempts of verifying larger models with symbolic model checking with SAT/SMT instead of BDDs has been presented in Biere et al. (1999a) and Clarke et al. (2001). They propose the usage of bounded model checking, which relies on the unrolling of the transition relations as a SAT program. Although it can solve more complex problems than BDD based, unbounded model checking (see Biere et al. (1999a)), it has one trade off - it only computes a bounded number of steps. Moreover, it is hard to compute if the number of steps cover the whole system behavior. For unbounded, BDD based model checking this is always the case.

Another approach of using SAT/SMT for model checking is the Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3), presented by Bradley (2012). With the help of induction and logical reasoning, it computes iteratively whether a hazardous state can be reached from an initial state. Therefore, it checks if there exists a trace between the hazardous and any initials state within n steps. Although, it can handle larger models than the BDD based approach, it also suffers from the state space explosion problem implying a high computation time for large state formulas.

Another part of our method is model based slicing. Originally, in the verification purpose, slicing was not developed for state models but for programs (Weiser (1981)). It was used to analyze program behavior with respect to some special variables and to find all lines of code influencing those variables.

First attempts to use slicing in the model verification world had been made by Clarke et al. (1999a) as well as Korel et al. (2003). Both showed that under special model conditions it is possible to reduce the resulting state space by slicing the model before verification. In general, they did the slicing based on the hardware description language that had been used, but they used slicing as a kind of preprocessing to eliminate transition rules, which have no influence on the given specification. In the end, they also use some BDD model checker to verify their model.

One other verification tool, where formal models are proven with the help of forward tracing and backtracking is KIV, presented by Haneberg et al. (2005). Its main idea is to execute possible proving traces until a dead end state or, in contrast to that, some goal state is reached. In the case a dead end is found, backtracking is executed to the last state where some decision had been required.

Summarizing, model based slicing can be used to find transition rules that have influence on a specific specification property. Moreover, encoding a transition system into a SAT/SMT program showed to be a useful alternative to BDDs that can be used for the encoding of transition rules for further research in model checking.

### 4. BACKWARD MODEL CHECKING WITH SLICING

This section describes the Backward Model Checking with Slicing approach. The basic idea is to decrease the impact of the state space explosion problem, i.e. the model size, on the verification process. From an abstract point of view, the algorithm can handle any transition based, finite state system. However, in this paper we restrict ourselves to models in the language SAML.

The core idea of the approach is to slice through the model with the help of the transition rules (instead of searching in the flattened state space). We think that this advantage can decrease the influence of the state space size on the model checking, because it does not require to compute the state space or some symbolic representation.

Starting from a hazardous state combination, i.e. a state vector not satisfying a given property $p$, the algorithm tries to slice backward through the state space to reach any initial state. Altogether, 6 steps are necessary for a meaningful evaluation.
• represent a set of states
• encode transition rules for backward slicing
• applying a transition rule
• finding suitable paths
• model checking a chosen path
• generating a counter example

4.1 Representing the Systems State with Abstract States

Although the conditions of the transition rules within a component are mutual exclusive, i.e., only one rule applies for each state, for the backward traversing it is not. So, if a transition rule is applied backward on a state it could happen that it results in a set of possible “previous” states. Imagine the following transition rule:

\[
\text{state} < 2 \rightarrow \text{state'} = 1;
\]

While slicing backward, this transition rule gets active if \( \text{state'} = 1 \). In this case, the possible previous state are those, which are less than 2. According to the given variable range this could be more than one but multiple states.

Due to this, the representation of a system state, as a valuation vector of each state variable, must be adjusted. Therefore, an abstract state \( \tilde{s} \) is defined, which represents not a single state, but a vector containing multiple values for each state variable.

Further, an abstract state is needed, because the algorithm typically does not start its search from one state, e.g. the initial state, but from a set of states defining a hazardous behavior. While initial states might often be unique, hazardous states are typically not (e.g. plane on ground with arbitrary speed but gears up).

Note that for the rest of this section we use primed variables \( \tilde{s}' \) to denote a state which could be derived as next state from some (unprimed) state \( \tilde{s} \) following the transition system. However, as we do backwards slicing inverted conditions are applied to primed variables and inverted assignments assign values to unprimed variables.

4.2 Encoding Transition Rules for Backward Slicing

As mentioned in the background, in SAML a transition rule \( t = (C, A) \) is a tuple of some condition rule \( C \) and an assignment rule \( A \). If the current state \( \tilde{s} \) satisfies the condition \( C \), the assignment \( A \) is applied, \( \tilde{s}' = A(\tilde{s}) \). The result is one step forward in the transition system, from a state \( \tilde{s} \rightarrow \tilde{s}' \) to a next state. Simplified, a transition rule defines a transition from a set of states into a set of next states. Very often transition rules describe multiple transitions (for example the transition describing the behavior of the water tank in listing 1 ll.19).

However, the goal of our algorithm is to step backward through the model’s states. Therefore, the transition rules have to be interpreted backward, i.e., a transition from \( \tilde{s}' \rightarrow \tilde{s} \). So, we need to invert a transition rule to get a backward transition rule.

\[
t^{-1} = (C^{-1}, A^{-1}) \quad (3)
\]

**Inverting a Condition:** The basic idea is to define a formula with only primed variables as free variables, which states that the primed state might be compatible with the current state \( \tilde{s}' \). In essence, this means to check whether the assignment of the transition rule under consideration is consistent with the current state or not. Thereby, a new condition \( C^{-1} \) is created. It defines the set of states, which could have been reached by applying this transition rule in forward direction, i.e., the set of the states \( \tilde{s}' \) that could have been reached with the transition rule. To do this, we rewrite the assignment into a Boolean formula \( C^{-1} \) containing both primed and unprimed variables.

\[
C^{-1}(\tilde{s}, \tilde{s}') := \tilde{s}' = A(\tilde{s})
\]

For checking consistency with a given state \( \tilde{s}' \), we check this formula in the context of \( \tilde{s} \) for satisfiability. This also often allows to simplify the SMT problems. In the example of listing 1, the inverse condition would be \( C^{-1} := faucet_\text{ctrl}' = open \land valve_\text{ctrl}' = closed \land true \).

Note that the second part of the initial assignment could be simplified to true as it does not place any restrictions on the primed state (i.e. the right side term of the assignment does neither involve constants nor primed variables). However, if unprimed variables (like in the second choice of this example) are used within the assignment, we will use them in generating the inverse assignment. Generating inverse conditions can be done once at the beginning and does not change.

**Inverting an Assignment** For computing, the next state set or more precise the previous state set an inverted assignments must be generated.

For the definition of the inversion of an assignment, two cases may be distinguished. If the assignment of the transition rule is (i) an assignment of some fixed value or (ii) an expression containing other state variables. The first situation is easy. The inverted assignment is then simply the condition of the original transition rule

\[
\tilde{s} \in A^{-1}(\tilde{s}') \iff C(\tilde{s}) = true.
\]

This means, any state \( \tilde{s} \) fulfilling the original condition \( C \) is a possible pre-decessor of state \( \tilde{s}' \). For example \( t : state = open \land x > 5 \rightarrow state' = closed \land true \). Inverted assignment \( A^{-1} \) such that \( state = open \land x > 5 \). Note, this assignment is typically not deterministic and might be an abstraction.

In contrast to that, if the assignment is (ii) an expression containing only constants and the current state variable (i.e state variable from which the next state is calculated), the whole expression \( term \) is taken as an assignment (in this context \( term(\tilde{s}) \) represents the function generating the value of \( \tilde{s}' \)).

\[
\tilde{s} \in A^{-1}(\tilde{s}') \iff C(\tilde{s}) \land (\tilde{s}' = term(\tilde{s}))
\]

In theory, to get a traditional assignment rule \( term \) would have to be inverted. This might be complex, however, in most practical applications the terms of an assignment are not to complicated.

4.3 Backwards Execution by SMT-Checking

Together, these encodings allow us to define some algorithm for backward execution using SMT checking. The algorithm is pretty simple and consists of three steps. The first step is to compute the abstract hazard state, i.e. the
entry state for our algorithm. Then iteratively predecessor states are computed using SMT checking and its produced solution models. Finally, it is checked whether an initial state is reached.

**Computing Hazardous States** Finding hazardous states is done according to CTL model checking and we thus do not go into detail here but refer to standard literature Clarke et al. (1999b), Clarke (1997) and Katoen and Baier (2008). The core idea is to unroll a CTL formula from inside out, i.e. from the most inner CTL operator to the top level one. As the only atomic constructs of CTL formula are atomic propositions, these can easily be identified with a boolean formula representing all fulfilling states. We use this boolean representation as initial state. In the case the formula does not cover all state formulas, for the free variables we take their initial bound as state value.

**Stepping backward** In the previous section, we described how inverted transition rules can be derived. Such a inverted rule can now be described as some simple SMT problem. This is done in a three step process.

**Step 1:** First, we compute possible applicable (inverted) transition rules per component. This is done by checking the inverted conditions of all transition rules per component. The big advantage is that this is done for each parallel component. The single check of each component brings a huge benefit as it allows for decomposition of the SMT problem.

In theory for each component an arbitrary number of (inverted) transition rules might exist. To determine which are actually applicable for some given state \( \tilde{s} \), would require a number of SMT checks equal to the number of transition rules. This would increase computation effort a lot. For this paper, we restrict ourselves to existential properties (i.e. the inversion of \( AG \phi \) properties). As a consequence, we focus on clever heuristics for finding one single path modeling \( \phi \). Therefore, we only need heuristics ranking transition rules per component by their possible execution order for the backward slicing. This ranking could be either done according to the users choice or we simply take the ordering given in the input model. The second heuristic is derived from the fact that very often we model transition systems by mentally mapping the evolution of the system to the transition rules. This means, we write down those transitions first, which start from the initial state and leads to the final state or some recurring. However, this is just some heuristics and in case we could not reach an initial state, we might need to backtrack anyway. The result of this step is one applicable transition rule per component under the condition of one given abstract primed state \( \tilde{s}' \).

**Step 2:** The next step is to compute previous (non primed) states. We do this by solving some SMT problem representing one applicable transition rule per component. The SMT problem is made of three elements:

(i) formulas specifying the domain/range of state variables,

(ii) formulas describing the current state and

(iii) formulas describing the inverted assignments of the transition rules.

**Domain/Range of State Variables** For efficient SMT checking, the set of available variables and their domains have to be announced. Since each state variable in some SAML model has a fixed domain range, this information is taken and yields the following rules:

\[
v_i \geq lower\text{Bound}(D_i) \land v_i \leq upper\text{Bound}(D_i)
\]

This ensures that any state variable \( v_i \) never gets a value assigned, which is out of its domain range. The function \( lower\text{Bound}(D_i) \) represents the lower bound of the state variable range and \( upper\text{Bound}(D_i) \) the upper bound of the range. For the set of all state variables \( V \) this results in the formula

\[
F_{domain} := \bigwedge_{0 \leq i \leq |V|} (v_i \geq lower\text{Bound}(D_i)) \land (v_i \leq upper\text{Bound}(D_i))
\]

**Current State** The current state is a valuation of the state variable \( v_i \). This can be easily described as a formula:

\[
F_{state} := \bigwedge_{0 \leq j \leq k} v_i = d_i,
\]

If the state is abstracted – i.e. more than one valuation is possible – it may be described by some disjunction of states or any other (minimized Boolean formula)

\[
F_{state} := \bigvee_{0 \leq i \leq n} \bigwedge_{0 \leq j \leq k} v_i = d_{i,j}.
\]

**Transition Rules** This is the really important part of the SMT problem. We already determined active transition rules in step 1. For computing the previous state \( \tilde{s} \) of a given state \( \tilde{s}' \), we need to take both the inverted assignment as well as the inverted condition into account.

\[
F_{transitions} := \bigvee_{0 \leq i < |Comp| \tilde{s} \in A^{-1}(s) \land (C^{-1}(\tilde{s}, \tilde{s}'))
\]

Note that \( C^{-1} \) or \( A^{-1} \) denote the condition or assignment of the inverted transition rule and not(!) the negation of the condition resp. assignment. Definitions for both have been given in Sect. 4.2.

Together this yields to the SMT problem:

\[
satisfiable := (F_{domain} \land F_{state} \land F_{transitions})
\]

Once this problem has been solved, we check if the a new state \( \tilde{s} \) (a computed predecessor state of \( \tilde{s}' \)) is an initial state. If not, we take the new state \( \tilde{s} \) as primed state \( \tilde{s}' \) and iterate.

**4.4 Backtracking during backwards execution**

Translating to decomposed SMT problems is highly efficient, however it also brings a number of problems, which might make backtracking necessary. One has already been mentioned in the previous section (e.g. if the wrong transition rules has been chosen). As a consequence, we could have chosen inconsistent transition rules. This might happen if for at least one component two transition rules are applicable for a given state \( \tilde{s} \) and the activation condition of the original transition rule is not compatible with the activation condition of another component’s transition rule. As an example, assume a system with two components \( C_1 \) and \( C_2 \). For a given state \( \tilde{s} \) further assume, two transition rules of \( C_1 \), e.g. \( TR_{A,C_1} \) and \( TR_{B,C_1} \), are applicable but
only one rule, \( T_{RA,C2} \), is applicable for component \( C_2 \). It could now arise the situation that our heuristics choose the pair \( \{ T_{RA,C1}, T_{RA,C2} \} \) for stepping backwards. However, if the activating conditions of \( T_{RA,C1} \) and \( T_{RA,C2} \) would be contradicting, the SMT problem of Sect. 4.3 would become unsatisfiable. In this case, we simply choose the second best applicable transition rule. Note, that this case seems to be very rare in practical application.

4.5 Backtracking outside backwards execution

Basically, the proposed algorithm follows two main ideas: (i) decomposing a large problem into several (smaller) SMT problems and (ii) following clever search heuristics. The main challenge is to decide how and when to backtrack. Backtracking (outside) might be necessary for two reasons. Either we chose the wrong transition rule or we would need another SMT solution of the problem of Sect. 4.3. The first could be done by simple "standard" backtracking, i.e. choose another applicable transition rule and repeat the calculation. The second is more tricky. The naïve approach – as described in the previous section – of simply listing states in disjunctive normal form does not scale well. However, some heuristics to generalize might be to simply weaken the state formula per component, i.e., to exclude all states, which had already been visited and did not lead to any initial state. This means to allow for multiple values inside the inner term of \( F_{\text{state}} \). e.g.

\[
F_{\text{state}} := \bigvee_{0 \leq i \leq n} \left( \bigwedge_{0 \leq j \leq k} \bigwedge_{0 \leq l \leq m} v_i = d_{i,j,k} \right).
\]

Thus, each individual state in this disjunction now represent some family of states, which are differing in only a number of state variables. This can of course be translated into disjunctive normal form again. Remember that individual state variables are defined in exactly one component. This allows for automatically finding abstractions much more easy. However, it is part of our current research to find generic abstraction mechanism for a state.

5. IMPLEMENTATION AND EVALUATION

5.1 Implementation

For the following evaluation, we needed to build a prototype. The prototype's structure is presented in Fig. 1. It consists mainly of three parts: The ModelBuilder and the Transition Rule Encoder, which translates the input model as well as the specification into the required SMT program, the Slicer, which is responsible for the backward model checking, i.e. the slicing through the state space, and the SMT Connector with the SMT solver, which are responsible for the SMT based computations. Additionally, the SMT solver has to implement the SMTLib standard 2.0 or higher (C. Barrett and Tinelli (2012)). For the prototype we used the SMTInterpol solver (Christ et al. (2012)). This is, because the prototype and SMTInterpol are both written in Java and SMTInterpol reached good evaluations in the 2014 SMTComp\(^4\) competition for SMT solvers.

\(^4\) http://smtcomp.sourceforge.net/2014/results-toc.shtml

Fig. 1. Structure of the prototype.

SMTLib2 Translation To be able to change the SMT-Solver, e.g. for evaluation purposes or to use the latest implementations, we choose the SMTLib2 specification as language for our transition rule encodings presented in the previous section. Hence, we are able to use every solver implementing the SMTLib2 standard. Listing 2 presents an exemplary encoding of a transition rule (Listing1 l. 7-11) into SMTLib2 with the formulas given in the previous section. From l. 1-3 the state variable \( \text{user.valve} \), \( \text{ctrl} \) is encoded. The primed version is encoded in l.7. Here we substitute the " ' " by "0". Line 10-26 represents the mentioned transition rule. It conjuncts the condition and the corresponding nondeterministic assignments. The nondeterministic behavior is encapsulated within a disjunction, because either the one or the other is sufficient to activate the transition rule. In line 29, the current value of the state variable \( \text{user.valve.ctrl} \), 0, is encoded.

1  :state variable user.valve.ctrl with 0:open
2  :and 1:closed
3  { declare-fun user.valve.ctrl() Int }
4  { assert (and (< user.valve.ctrl 1) (> user.valve.ctrl 0)) }
5  { primed state variable user.valve.ctrl0() Int }
6  { assert :encoded transition rule
7  (and
8  :condition C
9  (< tank..level 10)
10  { or
11  { assignments A
12  { and
13  (= user.valve.ctrl1 1)
14  (= user.valve.ctrl0 0)
15  )
16  (and
17  (= user.valve.ctrl0 user.valve.ctrl)
18  (= user.faucet.ctrl1 user.faucet.ctrl)
19  )
20  )
21  :current state
22  { assert (= user.valve.ctrl0 0))
Listing 2. Water tank example as SMTLib model.

The presented SMTLib2 code can be used to check whether some transition rule is applicable with respect to
a specific system state. To calculate the previous state, we put all transition rule encodings, state variables and the current state into one program. If the verification result of the used STM-Solver is SAT for satisfiable, we take the representative model as the next previous state.

5.2 Test Setup and Models

For the evaluation, we used some standard PC (Intel Core i7-3930K@3.20GHz, Ubuntu 14.04, RAM 64GB). As benchmark, we used muXmv (version 1.0.1). For generating the muXmv model, we used the automatic model transformations, which are part of the VECS- framework. A short overview of VECS is presented by Gonschorek et al. (2014).

Due to the early stage of our approach, we restrict ourselves to proofs of CTL formulas of the type $AG\phi$, which means on all paths in all states $\phi$ must hold. Extending our approach to other types of CTL formulas would have to be done in a similar way like unrolling of CTL formula for model checking is done. However, this is ongoing work.

Finding representative test cases was a difficult question. The requirements for a test case are that (a) it requires significant verification effort for muXmv and (b) we could deduce the complexity into the model to interpret the results. As a solution we chose to define a set of test models that were of similar complexity (in terms of BDD size) but rising in difficulty for our approach. The "best" models for our approach are counter-like automata with few, perfectly ordered transition rules. As a consequence our heuristic would never need to back track and find the initial state after exactly $n$-iterations (if the hazardous state took $n$-iterations to reach). The second class of models were designed, such that backtracking within one step was necessary. For the third class, we also introduce backtracking of multiple steps. This corresponds to situations where ordering of the transition rules was not perfect initially.

Finally, the last class, class 4, of test cases also included backtracking in the solutions states (i.e. the SMT solver’s initial solutions where not the ones that really lead to the initial state).

### Class 1
Test model 1, 2 and 3 are simple counter with one state variable and two transition rule of the form:

\[
state:\{0..100000\} \init 0;
state < 100000 \rightarrow state' = state + 1;
state = 100000 \rightarrow state' = state;
\]

Only the state variable range varies over 10,000 (Model 1), 100,000 (Model 2) and 1,000,000 (Model 3). The corresponding specification is $AG\ state < [maxVariableRange]$.

### Class 2
Test model 4 and 5 extends the model with a partition of the transition rules, i.e. the first transition rule is split into 10 (Model 4) and 100 (Model 5) transition rules whereas each covers a consecutive rang of the state values, e.g. from 1,000 to 2,000. The overall variable range is 100,000.

### Class 3
For test model 6, nondeterministic behavior had been added to construct loops in the automaton. This is done by adding $state'=0$ to the possible assignments of the transition rules.

<table>
<thead>
<tr>
<th>Testmodel</th>
<th>BDD</th>
<th>BWMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Class 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testmodel 1</td>
<td>6s</td>
<td>16s</td>
</tr>
<tr>
<td>Testmodel 2</td>
<td>612s</td>
<td>76s</td>
</tr>
<tr>
<td>Testmodel 3</td>
<td>&gt;10.800s (3h)</td>
<td>675s</td>
</tr>
<tr>
<td>Test Class 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testmodel 4</td>
<td>683s</td>
<td>72s</td>
</tr>
<tr>
<td>Testmodel 5</td>
<td>725s</td>
<td>71s</td>
</tr>
<tr>
<td>Test Class 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testmodel 6</td>
<td>656s</td>
<td>72s</td>
</tr>
<tr>
<td>Testmodel 7</td>
<td>94s</td>
<td>51s</td>
</tr>
</tbody>
</table>

Table 1. Test Results

\[
\text{state} < 1000000 \rightarrow choice:\{(state' = state + 1) + (state' = 0)\};
\]

Here, the variable range is also 100,000 and the number of transition rules is 100. For test model 6 we have switched from one component to ten parallel component of which each counts from 0 to 10,000 in parallel. For this test model the specification was that $AG\ compl.\ state < 10000$. We have taken this one because it implies that our algorithm start with 9 free variables, which leads to the case that there is not only backtracking in transition rules but also in variables.

5.3 Results

In Table 5.3 the test results are presented. The first column presents the BDD based verification with muXmv and the second column shows the results for the BWMC approach. In all test models, the verification found a trace between initial and hazardous state, if the program execution has not been interrupted. But this had only been the case for test model 6, because muXmv took more than 3 hours for the computation.

One interesting fact given by the results is that in six of seven tests BWMC is faster than the BDD approach. Only the smallest model (test model 1) had been solved faster by muXmv. Another point is that the computation time of the BWMC approach increases nearly linear to the state space (test model 1, 2, 3), whereas the computation time of the BDD approach seems to increases quadratically.

Moreover, there is no influence of backtracking within the transition rules (model 4 and 5) for the BWMC approach, but for the BDD one. However, this difference is more extreme.

Test model 6 shows that the influence of simple loops is rather small, for both BWMC and BDD.

Further, the BDD approach had been taken more time to compute the 10 parallel components than the BWMC approach. Also the multiplication factor for 1 component with 10,000 states to 10 components with 10,000 states each is higher for the BDD (ca. $\times 15$) as for the BMWC (ca. $\times 3.5$) approach.
Note, that these results are preliminary. On the one hand, our implementation using SMTLib is still not optimal. On the other hand, we are of course aware that the real challenge is to take realistic models for a comparison. This means models, which were not defined for a benchmark situation but rather from some real application.

6. CONCLUSION

In this paper, we presented an algorithm for explicit backward model checking. The idea itself is not new. However, we believe that a big potential can be developed by defining search heuristics interpreting the structure of the model. The core idea relies on the assumption that "good" heuristics can be defined by users on specification level – potentially even implicitly. The second main ingredient is to make use of efficient SMT solving technologies for finding good and fast previous states. The main challenge here will be to find the right trade-off between generality (coding all assignments into SMT problems) and specific optimizations (defining inverse operations for some frequently used assignments). Our small evaluation shows that there is a potential to solve larger models than current BDD model checkers are able to. Of course, we have to evaluate larger and more realistic models, but the chosen ones show that Backward Model Checking with Slicing can be an usable approach in future applications. Finally, it will have to be evaluated to what extent the possible decomposition of backtracking problems into sub-problems really helps in realistic models.

REFERENCES