The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

August 10 - 22, 2006

A CAV'06 Workshop

BMC'06: Bounded Model Checking

August 15st, 2006

Proceedings

Editors:

Armin Biere, Ofer Strichman

BMC'06

Fourth International Workshop on Bounded Model Checking

Preliminary Proceedings

August 15, 2006

Seattle, Washington, USA

Preface

These are the *preliminary proceedings* of the fourth international workshop on Bounded Model Checking (BMC'06) that was held on Aug 15th, 2006 in Seattle, Washington, USA as a CAV'06 affiliated workshop (also part of FLOC'06). The final proceedings will be published in Electronic Notes in Theoretical Computer Science (ENTCS), together with other Computer Aided Verification (CAV'06) workshops. All the 5 submissions were chosen to be included in these proceedings and workshop, although some of them will go through another round of reviews before publishing them in the official proceedings. Each of these papers was reviewed by three program committee members. The workshop had two invited talks: one by Ilkka Niemela about 'Bounded Model Checking, Answer Set Programming, and Fixed Points' and one by Fabio Somenzi about 'Techniques for proving properties with SAT-based MC'. The first session of the workshop was a joint session with SAT'06 (not listed in the following program).

We thank the program committee for their effort in evaluating the articles and giving helpful comments to the authors. We also thank the organizers of the hosting conference, CAV'06, Thomas Ball and Robert Jones.

Organizers

Armin Biere	Ofer Strichman
Johannes Kepler University, Linz, Austria	*Technion, Haifa, Israel*

Program Committee

Per Bjesse, *Synopsys, USA*	Alan Hu, *UBC, Canada*
Alessandro Cimatti, *IRST, Italy*	Yunshan Zhu, *Consultant, USA*
Koen Claessen, *Chalmers, Sweden*	Aarti Gupta,*NEC, USA*
Ranan Fraer, *Intel, Israel*	Ken McMillan, *Cadence, USA*
Danny Geist,*Intel, Israel*	Daniel Kroening, *ETH-Zurich, Switzerland*
João Marques Silva, *University of Southampton, UK*	

Armin Biere, Ofer Strichman — Linz, Haifa, June 2006

Contents

Invited Talk

Bounded Model Checking, Answer Set Programming, and Fixed Points

Ilkka Niemela
Helsinki University of Technology Dept. of Computer Science and Engineering Laboratory for Theoretical Computer Science, Helsinki, Finland
Ilkka.Niemela@tkk.fi

Abstract
Abstract: Answer set programming (ASP) is a novel declarative paradigm for solving search problems. The basic idea of ASP is similar to, for example, SAT-based planning or constraint satisfaction problems but ASP has its roots in logic programming and the stable model semantics. ASP offers a powerful knowledge representation language for effective problem encoding supporting, for example, logical variables, recursive definitions, and a variety of aggregates. A number of successful ASP systems have already been developed and applied in areas such as planning, decision support for the flight controllers of space shuttles, web-based product configuration, configuration of a Linux distribution, VLSI routing, and linguistics.

The talk explains the theoretical underpinnings of ASP, outlines computational techniques used in current ASP solvers, and then discusses bounded model checking (BMC) as a potential application area for ASP techniques. BMC can be seen as a search problem where the task is to find an execution of the system violating a given (temporal) property. ASP fit well such an application and offers promising techniques to encoding and solving BMC problems. In particular, ASP supports directly recursive fixed point computation needed to evaluate temporal operators and we discuss how this can be used in bounded model checking of linear temporal logic (LTL).

Towards an Efficient Path-Oriented Tool for Bounded Reachability Analysis of Linear Hybrid Systems using Linear Programming

Xuandong Li [a,b,1], Sumit Jha Aanand [b,2] and Lei Bu [a,3]

[a] *State Key Laboratory of Novel Software Technology*
Department of Computer Science and Technology, Nanjing University
Nanjing, Jiangsu, P.R.China 210092

[b] *Computer Science Department, Carnegie Mellon University*
5000 Forbes Avenue, Pittsburgh, PA15213, USA

Abstract

The existing techniques for reachability analysis of linear hybrid automata do not scale well to problem sizes of practical interest. Instead of developing a tool to perform reachability check on all the paths of a linear hybrid automaton, a complementary approach is to develop an efficient path-oriented tool to check one path at a time where the length of the path being checked can be made very large and the size of the automaton can be made large enough to handle problems of practical interest. This approach of symbolic execution of paths can be used by design engineers to check important paths and thereby, increase the faith in the correctness of the system. Unlike simple testing, each path in our framework represents a dense set of possible trajectories of the system being analyzed. In this paper, we develop the linear programming based techniques towards an efficient path-oriented tool for the bounded reachability analysis of linear hybrid systems.

Key words: Linear hybrid automata, bounded model checking, reachability analysis, linear programming.

1 Introduction

The model checking problem for hybrid systems is very difficult. Even for a relatively simple class of hybrid systems - the class of linear hybrid automata [1] - the most common problem of reachability is still undecidable [1-3].

[1] Email: `lxd@nju.edu.cn`
[2] Email: `jha+@cs.cmu.edu`
[3] Email: `bl@seg.nju.edu.cn`

Several model checking tools have been developed for linear hybrid automata, but they do not scale well to the size of practical problems. The state-of-the-art tool HYTECH [8] and its improvement PHAVer [9] need to perform expensive polyhedra computation, and their algorithm complexity is exponential in number of variables in the automata. In recent years, the bounded model checking has been presented as a complementary technique for BDD-based symbolic model checking, whose basic idea is to search for a counterexample in the model executions whose length is bounded by some integer k [5]. Several works [6,7] have been given to check linear hybrid systems using the bounded model checking technique. In these techniques, the model checking problems are reduced into the satisfiability problem of a boolean combination of propositional variables and mathematical constraints, but their experiment results show that the length of the checked model executions is still far from the practical problem size.

As the existing techniques cannot check all the paths for reachability analysis when attempting analysis of problem sizes that are of practical significance, a complementary approach is to develop an efficient path-oriented tool to check one path at a time where the length of the path being checked can be made very large and the size of the automaton can be made large enough to handle problems of practical interest. This approach of symbolic execution of paths can be used by the design engineers to check important paths and thereby, increase the faith in the correctness of the system. Unlike simple testing, each path in our framework represents a dense set of possible trajectories of the system being analyzed. In this paper, we present the linear programming based techniques towards development of an efficient path-oriented tool for the bounded reachability analysis of linear hybrid systems.

The paper is organized as follows. In next section, we define the class of linear hybrid automata considered in this paper. In section 3, we use linear programming to present our solution for the path-oriented bounded reachability analysis of linear hybrid automata. Section 4 presents some techniques to reduce the size of the linear programs corresponding to the paths that we are checking. The tool prototype and the case studies are described in section 5 . We give the conclusion in the last section.

2 Linear Hybrid Automata

The linear hybrid automata considered in this paper are a variation of the definition given in [1], in which the change rates of variables may be given a range of possible values. For simplicity, we suppose that in any linear hybrid automaton, considered in this paper, there is just one initial location with no initial conditions and no transitions to the initial location (we assume that each variable with an initial value is reset to the initial value by the transitions from the initial location).

Definition 2.1 A linear hybrid automaton is a tuple $H = (X, V, E, v_I, \alpha, \beta)$, where

- X is a finite set of real-valued variables. V is a finite set of *locations.*
- E is *transition* relation whose elements are of the form (v, ϕ, ψ, v') where v, v' are in V, ϕ is a set of *variable constraints* of the form $a \leq \sum_{i=0}^{m} c_i x_i \leq b$, and ψ is a set of *reset actions* of the form $x := c$ where $x_i \in X$ $(0 \leq i \leq m)$, $x \in X$, a, b, c and c_i $(0 \leq i \leq m)$ are real numbers, and a and b may be ∞.
- v_I is an *initial* location.
- α is a labelling function which maps each location in $V - \{v_I\}$ to a *state invariant* which is a set of variable constraints of the form $a \leq \sum_{i=0}^{m} c_i x_i \leq b$ where $x_i \in X$ $(0 \leq i \leq m)$, a, b, and c_i $(0 \leq i \leq m)$ are real numbers, a and b may be ∞.
- β is a labelling function which maps each location in $V - \{v_I\}$ to a set of *change rates* which are of the form $\dot{x}= [a, b]$ where $x \in X$, and a, b are real numbers $(a \leq b)$. For any location v, for any $x \in X$, there is one and only one change rate definition $\dot{x}= [a, b] \in \beta(v)$. $\square$

Notice that the class of linear hybrid automata we consider here can be used to approximate any general hybrid automata to any desired level of accuracy because they are sufficiently expressive to allow asymptotically completeness of the abstraction process for a general hybrid automata [4].

We use the sequences of locations to represent the evolution of a linear hybrid automaton from location to location. For a linear hybrid automaton $H = (X, V, E, v_I, \alpha, \beta)$, a *path segment* is a sequence of locations of the form

$$v_1 \xrightarrow{(\phi_1,\psi_1)} v_2 \xrightarrow{(\phi_2,\psi_2)} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n$$

which satisfies $(v_i, \phi_i, \psi_i, v_{i+1}) \in E$ for each i $(1 \leq i \leq n-1)$. A *path* in H is a path segment starting at v_I.

The behavior of linear hybrid automata can be represented by *timed sequences.* Any timed sequence is of the form $(v_1, t_1)\hat{}(v_2, t_2)\hat{}\ldots\hat{}(v_n, t_n)$ where v_i $(1 \leq i \leq n)$ is a location and t_i $(1 \leq i \leq n)$ is a nonnegative real number. It represents a behavior of an automaton, that is, the system starts at the initial location and changes to the location v_1, stays there for t_1 time units, then changes to the location v_2 and stays at v_2 for t_2 time units, and so on.

Definition 2.2 For a linear hybrid automaton $H = (X, V, E, v_I, \alpha, \beta)$, a timed sequence $(v_1, t_1)\hat{}(v_2, t_2)\hat{}\ldots\hat{}(v_n, t_n)$ represents a behavior of H if and only if the following condition is satisfied:

- there is a path in H of the form $v_0 \xrightarrow{(\phi_0,\psi_0)} v_1 \xrightarrow{(\phi_1,\psi_1)} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n$;
- $t_1, t_2, \ldots, t_n$ satisfy all the variable constraints in ϕ_i $(1 \leq i \leq n-1)$, i.e. for each variable constraint $a \leq c_0x_0 + c_1x_1 + \ldots + c_mx_m \leq b$ in ϕ_i,

$$\delta_k \leq \gamma_i(x_k) \leq \delta'_k \text{ for any } k \ (0 \leq k \leq m), \text{ and}$$

$$a \leq c_0\gamma_i(x_0) + c_1\gamma_i(x_1) + \ldots + c_m\gamma_i(x_m) \leq b$$

where $\gamma_i(x_k)$ $(0 \le k \le m)$ represents the value of the variable x_k when the automaton stays at v_i with the delay t_i, and for any k $(0 \le k \le m)$,

$$\delta_k = d_k + u_{j_k+1}t_{j_k+1} + u_{j_k+2}t_{j_k+2} + \ldots + u_i t_i,$$

$$\delta'_k = d_k + u'_{j_k+1}t_{j_k+1} + u'_{j_k+2}t_{j_k+2} + \ldots + u'_i t_i,$$

$x_k := d_k \in \psi_{j_k}$ $(0 \le j_k < i)$, $x_k := d \notin \psi_l$ for any l $(j_k < l < i)$, and $\dot{x}_l = [u_l, u'_l] \in \beta(v_l)$ for any l $(j_k < l \le i)$; and

- $t_1, t_2, \ldots, t_m$ satisfy the state invariant for each location v_i $(1 \le i \le n)$, i.e.

  · for each variable constraint $a \le c_0x_0 + c_1x_1 + \ldots + c_mx_m \le b$ in $\alpha(v_i)$,

  $$\delta_k \le \gamma_i(x_k) \le \delta'_k \text{ for any } k \ (0 \le k \le m), \text{ and}$$

  $$a \le c_0\gamma_i(x_0) + c_1\gamma_i(x_1) + \ldots + c_m\gamma_i(x_m) \le b$$

  where $\gamma_i(x_k)$ $(0 \le k \le m)$ represents the value of the variable x_k when the automaton stays at v_i with the delay t_i, and for any k $(0 \le k \le m)$,

  $$\delta_k = d_k + u_{j_k+1}t_{j_k+1} + u_{j_k+2}t_{j_k+2} + \ldots + u_i t_i,$$

  $$\delta'_k = d_k + u'_{j_k+1}t_{j_k+1} + u'_{j_k+2}t_{j_k+2} + \ldots + u'_i t_i,$$

  $x_k := d_k \in \psi_{j_k}$ $(0 \le j_k < i)$, $x_k := d \notin \psi_l$ for any l $(j_k < l < i)$, and $\dot{x}_l = [u_l, u'_l] \in \beta(v_l)$ for any l $(j_k < l \le i)$; and

  · for each variable constraint $a \le c_0x_0 + c_1x_1 + \ldots + c_mx_m \le b$ in $\alpha(v_{i+1})$,

  $$\delta_k \le \gamma_i(x_k) \le \delta'_k \text{ for any } k \ (0 \le k \le m), \text{ and}$$

  $$a \le c_0\lambda_i(x_0) + c_1\lambda_i(x_1) + \ldots + c_m\lambda_i(x_m) \le b$$

  where $\gamma_i(x_k)$ $(0 \le k \le m)$ represents the value of the variable x_k when the automaton stays at v_i with the delay t_i, if $x_k := e_{i_k} \in \psi_i$ $(0 \le k \le m)$ then $\lambda_i(x_k) = e_{i_k}$ else $\lambda_i(x_k) = \gamma_i(x_k)$, and for any k $(0 \le k \le m)$,

  $$\delta_k = d_k + u_{j_k+1}t_{j_k+1} + u_{j_k+2}t_{j_k+2} + \ldots + u_i t_i,$$

  $$\delta'_k = d_k + u'_{j_k+1}t_{j_k+1} + u'_{j_k+2}t_{j_k+2} + \ldots + u'_i t_i,$$

  $x_k := d_k \in \psi_{j_k}$ $(0 \le j_k < i)$, $x_k := d \notin \psi_l$ for any l $(j_k < l < i)$, and $\dot{x}_l = [u_l, u'_l] \in \beta(v_l)$ for any l $(j_k < l \le i)$. □

3 Path-Oriented Bounded Reachability Analysis using Linear Programming

In this section we use linear programming to present a solution for the path-oriented bounded reachability analysis of linear hybrid automata.

3.1 Path-Oriented Bounded Reachability

For a linear hybrid automaton H, a reachability specification consists of a location v in H and a set φ of variable constraints, denoted by $\mathcal{R}(v, \varphi)$. We are concerned with the problem of checking whether a path in H satisfies a given reachability specification. The formal definition is presented below.

Definition 3.1 Let $H = (X, V, E, v_I, \alpha, \beta)$ be a linear hybrid automaton, and $\mathcal{R}(v, \varphi)$ be a reachability specification. A path ρ in H of the form

$$v_0 \xrightarrow{(\phi_0,\psi_0)} v_1 \xrightarrow{(\phi_1,\psi_1)} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n$$

satisfies $\mathcal{R}(v, \varphi)$ if and only if the following condition holds:

- $v_n = v$, and
- there is a behavior of H of the form $(v_1, t_1)\hat{}(v_2, t_2)\hat{} \ldots \hat{}(v_n, t_n)$ such that any variable constraint in φ is satisfied when the automaton stays at v_n with the delay t_n, i.e. for each variable constraint $a \le c_0x_0+c_1x_1+\ldots+c_mx_m \le b$ in φ,

  $$\delta_k \le \gamma_n(x_k) \le \delta'_k \text{ for any } k \ (0 \le k \le m), \text{ and}$$

  $$a \le c_0\gamma_n(x_0) + c_1\gamma_n(x_1) + \ldots + c_m\gamma_n(x_m) \le b$$

  where $\gamma_n(x_k)$ $(0 \le k \le m)$ represents the value of the variable x_k when the automaton stays at v_n with the delay t_n, and for any k $(0 \le k \le m)$,

  $$\delta_k = d_k + u_{i_k+1}t_{i_k+1} + u_{i_k+2}t_{i_k+2} + \ldots + u_nt_n\,,$$

  $$\delta'_k = d_k + u'_{i_k+1}t_{i_k+1} + u'_{i_k+2}t_{i_k+2} + \ldots + u'_nt_n\,,$$

  $x_k := d_k \in \psi_{i_k}$ $(0 \le i_k < n)$, $x_k := d \notin \psi_j$ for any j $(i_k < j < n)$, and $\dot{x}_j = [u_j, u'_j] \in \beta(v_j)$ for any j $(i_k < j \le n)$. $\square$

3.2 Representation of a long path

Since our tool is designed to check a path which is as long as desired and can handle linear hybrid automata of practical problem size, we first need to represent such a long path.

For a linear hybrid automaton $H = (X, V, E, v_I, \alpha, \beta)$, we can represent a path segment ρ in H of the form

$$v_0 \xrightarrow{(\phi_0,\psi_0)} v_1 \xrightarrow{(\phi_1,\psi_1)} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n$$

by a simple form $v_0\hat{}v_1\hat{} \ldots \hat{}v_n$, which is called *simple regular expression.* A *simple regular expression* (SRE) R and the path segment $\mathcal{L}(R)$ it represents are defined recursively as follows:

- if $v \in V$, then v is a SRE, and $\mathcal{L}(v) = v$;
- if R_1 and R_2 are SREs and there is a transition in E from the last location in $\mathcal{L}(R_1)$ to the first location in $\mathcal{L}(R_2)$, then $R_1\hat{}R_2$ is a SRE, and

  $$\mathcal{L}(R_1\hat{}R_2) = \mathcal{L}(R_1) \xrightarrow{(\phi,\psi)} \mathcal{L}(R_2)\,;$$

- if R is a SRE and there is a transition in E from the last location in $\mathcal{L}(R)$ to the first location in $\mathcal{L}(R)$, then R^k is a SRE where $k \ge 2$ is an integer, and

$$\mathcal{L}(R^k) = \underbrace{\mathcal{L}(R) \xrightarrow{(\phi,\psi)} \mathcal{L}(R) \xrightarrow{(\phi,\psi)} \dots \xrightarrow{(\phi,\psi)} \mathcal{L}(R)}_{k} .$$

Using the above definition, we can represent a long path to be checked as a SRE, and the SREs can be used as a text language for the input of the tool.

3.3 Reducing the Bounded Reachability Problems into Linear Programs

Now we show how the problem of checking a path for a given reachability specification can be reduced to a linear program.

Let $H = (X, V, E, v_I, \alpha, \beta)$ be a linear hybrid automaton, $\mathcal{R}(v, \varphi)$ be a reachability specification, and ρ be a path in H of the form

$$v_0 \xrightarrow{(\phi_0,\psi_0)} v_1 \xrightarrow{(\phi_1,\psi_1)} \dots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n$$

where $v_n = v$. For any timed sequence of the form $(v_1, t_1)\hat{}(v_2, t_2)\hat{}\dots\hat{}(v_n, t_n)$, if it is such that ρ satisfies $\mathcal{R}(v, \varphi)$, then the following condition must hold:

- $t_1, t_2, \dots, t_n$ satisfy all the variable constraints in $\phi_i (0 \le i \le n)$,
- $t_1, t_2, \dots, t_n$ satisfy all the variable constraints in $\alpha(v_i)$ $(1 \le i \le n)$, and
- $t_1, t_2, \dots, t_n$ satisfy all the variable constraints in φ,

which form a group of linear inequalities on $t_1, t_2, \dots, t_n$ (see Definition 2.2 and 3.1), denoted by $\Theta(\rho, \mathcal{R}(v, \varphi))$. It follows that we can check if ρ satisfies $\mathcal{R}(v, \varphi)$ by checking if the group $\Theta(\rho, \mathcal{R}(v, \varphi))$ of linear inequalities has a solution, which can be solved by linear programming.

In addition to $t_1, t_2, \dots, t_n$, each $\gamma_i(x_k)$ in Definition 2.2 and 3.1 also becomes a variable in the linear program corresponding to checking of a path. Notice that if the change rate of x_k is a constant ($\dot{x}_k = [a, a]$), then $\delta_k = \delta'_k$ in Definition 2.2 and 3.1 such that we can replace $\gamma_i(x_k)$ with δ_k. Thus, for a path checking, the numbers of the variables and the constraints in the corresponding linear program can be calculated as follows:

- we have one variable in the linear program for each location in the path,
- we have at most one variable in the linear program for each variable occurrence in a variable constraint labelled on a transition, in a location invariant, and in the reachability specification,
- for each variable occurrence in a variable constraint labelled on a transition, in a location invariant, and in the reachability specification, we have at most one constraint in the linear program,
- for each variable constraint labelled on a transition, we have one constraint in the linear program,
- for each variable constraint in a location invariant, we have two constraints in the linear program, and
- for each variable constraint in the reachability specification, we have one

constraint in the linear program.

Thanks to the advances in computing during the past decade, linear programs in a few thousand variables and constraints are nowadays viewed as "small". Problems having tens or hundreds of thousands of continuous variables are regularly solved. Indeed, many software packages have been developed to efficiently find solutions for linear programs. Leveraging the research in efficient solution of linear programs, we can develop an efficient tool to check a path in a linear hybrid automaton, where the length of the path and the size of the linear hybrid automaton are both closer to the practical problem sizes.

4 Reducing Size of Linear Programs Corresponding to Path Checking

We have reduced the bounded reachability analysis for a given path into a linear programming problem. In this section, we present several techniques for reducing the size of the resulting linear programming problem so that our tool can be used to solve problems of size as large as possible.

4.1 Decomposing Linear Programs Corresponding to Path Checking

In some cases, we can decompose the linear program corresponding to the path being checked into several smaller linear programs so that the tool can check longer paths.

Let $H = (X, V, E, v_I, \alpha, \beta)$ be a linear hybrid automaton, $\mathcal{R}(v, \varphi)$ be a reachability specification, and ρ be a path in H of the form

$$v_0 \xrightarrow{(\phi_0,\psi_0)} v_1 \xrightarrow{(\phi_1,\psi_1)} \ldots \xrightarrow{(\phi_{i-1},\psi_{i-1})} v_i \xrightarrow{(\phi_i,\psi_i)} v_{i+1} \xrightarrow{(\phi_{i+1},\psi_{i+1})} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n$$

where $v_n = v$. If there is i $(0 < i < n)$ such that

- for any variable x occurring in a variable constraint in ϕ_j $(i < j < n)$, x is reset on a transition $(v_k, \phi_k, \psi_k, v_{k+1})$ $(i \le k < j)$, i.e. $x := a \in \psi_k$,
- for any variable x occurring in a variable constraint in $\alpha(v_j)$ $(i < j \le n)$, x is reset on a transition $(v_k, \phi_k, \psi_k, v_{k+1})$ $(i \le k < j)$, i.e. $x := a \in \psi_k$, and
- for any variable x occurring in a variable constraint in φ, x is reset on a transition $(v_k, \phi_k, \psi_k, v_{k+1})$ $(i \le k < n)$, i.e. $x := a \in \psi_k$,

then the linear program corresponding to checking ρ for $\mathcal{R}(v, \varphi)$ can be decomposed. In this case, there is a timed sequence of the form

$$(v_1, t_1)\hat{}(v_2, t_2)\hat{}\ldots\hat{}(v_i, t_i)\hat{}(v_{i+1}, t_{i+1})\hat{}(v_{i+2}, t_{i+2})\hat{}\ldots(v_n, t_n)$$

such that ρ satisfies $\mathcal{R}(v, \varphi)$ if and only if the following condition holds:

- $t_1, t_2, \ldots, t_i$ satisfy all variable constraints in ϕ_k $(1 \le k \le i)$, and all variable constraints in $\alpha(v_k)$ $(1 \le k \le i)$, and

- $t_{i+1}, t_{i+2}, \ldots, t_n$ satisfy all variable constraints in ϕ_k $(i < k \leq n)$, all variable constraints in $\alpha(v_k)$ $(i < k \leq n)$, and all variable constraints in φ,

which correspond to two separate linear programs according to Definition 2.2 and 3.1. Thus, in this case we can decompose the linear program corresponding to a path checking into two smaller linear programs. The resulting linear programs can be recursively decomposed by the same technique until the technique can no longer be applied.

4.2 Shortening Paths

For a path segment ρ in a linear hybrid automaton, its *length* $|\rho|$ is the number of the locations in ρ. Since the size of the linear program corresponding to the path being checked is proportional to the length of the path, shortening the path will improve the complexity of the overall method. By shortening a path, we mean to find a shorter path in lieu of the path being checked such that both of them are equivalent with respect to the given reachability specification - if one of them satisfies the reachability specification, so does the other.

For a linear hybrid automaton $H = (X, V, E, v_I, \alpha, \beta)$, a long path ρ in H, which we want to check, usually includes repetitions of path segments, which can be represented as the following form:

$$\rho = v_0 \xrightarrow{(\phi_0,\psi_0)} \ldots \xrightarrow{(\phi_{i-1},\psi_{i-1})} v_i \xrightarrow{(\phi_i,\psi_i)} \rho_1^k \xrightarrow{(\phi,\psi)} v_{i+1} \xrightarrow{(\phi_{i+1},\psi_{i+1})} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n$$

where ρ_1 is a path segment in H, $k \geq 2$ is an integer, and ρ_1^k represents the path segment

$$\underbrace{\rho_1 \xrightarrow{(\phi',\psi')} \rho_1 \xrightarrow{(\phi',\psi')} \ldots \xrightarrow{(\phi',\psi')} \rho_1}_{k} .$$

In the following, we show that in some cases we can find $k' < k$ such that ρ satisfies a given reachability specification if and only if ρ' satisfies the reachability specification where ρ' is of the form

$$\rho' = v_0 \xrightarrow{(\phi_0,\psi_0)} \ldots \xrightarrow{(\phi_{i-1},\psi_{i-1})} v_i \xrightarrow{(\phi_i,\psi_i)} \rho_1^{k'} \xrightarrow{(\phi,\psi)} v_{i+1} \xrightarrow{(\phi_{i+1},\psi_{i+1})} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n .$$

Let $H = (X, V, E, v_I, \alpha, \beta)$ be a linear hybrid automaton, $\mathcal{R}(v, \varphi)$ be a reachability specification, and ρ be a path in H of the form

$$v_0 \xrightarrow{(\phi_0,\psi_0)} v_1 \xrightarrow{(\phi_1,\psi_1)} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n$$

where $v_n = v$. We say that a variable constraint is *related to* a location v_i $(0 \leq i \leq n)$ if it is in ϕ_i, $\alpha(v_i)$, or in φ when $i = n$. We define the *reference point* for a variable in a variable constraint related to a location in ρ as follows:

- for a variable x in a variable constraint related to a location v_i $(0 \leq i \leq n)$, a location v_j $(0 \leq j < i)$ is the *reference point* if x is reset on the transition $(v_j, \phi_j, \psi_j, v_{j+1})$ $(x := a \in \psi_j)$, and is not reset on any transition $(v_k, \phi_k, \psi_k, v_{k+1})$ $(j < k < i)$ $(x := b \notin \psi_k)$ (in this case, we say that a is the *reference value* of x on v_i).

Let $H = (X, V, E, v_I, \alpha, \beta)$ be a linear hybrid automaton, $\mathcal{R}(v, \varphi)$ be a reachability specification, and ρ be a path in H of the form $\rho = \rho_1 \xrightarrow{(\phi,\psi)} \rho_2^k \xrightarrow{(\phi',\psi')} \rho_1'$ where $k > 3$, ρ_1 is a path, and ρ_1', ρ_2 are path segments. We say that ρ_2^k is *closed* in ρ if the following condition holds:

- $\rho_2^k = \rho_{21} \xrightarrow{(\phi'',\psi'')} \rho_3^{k-2} \xrightarrow{(\phi'',\psi'')} \rho_{21}'$ where $\rho_2^2 = \rho_{21} \xrightarrow{(\phi'',\psi'')} \rho_{21}'$ and $|\rho_{21}| \leq |\rho_{21}'|$, and
- $\rho_3 = v_1 \xrightarrow{(\phi_1,\psi_1)} v_2 \xrightarrow{(\phi_2,\psi_2)} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n$, and for any x occurring in a variable constraints in ϕ_i or $\alpha(u_i)$ $(1 \leq i \leq n)$, $x := a \in \psi''$ or $x := a \in \psi_j$ $(1 \leq j < i)$.

Theorem 4.1 Let $H = (X, V, E, v_I, \alpha, \beta)$ be a linear hybrid automaton, $\mathcal{R}(v, \varphi)$ be a reachability specification, and $\rho = \rho_1 \xrightarrow{(\phi,\psi)} \rho_2^k \xrightarrow{(\phi',\psi')} \rho_1'$ be a path in H where ρ_2^k $(k > 3)$ is closed in ρ. If any location in ρ_1 is not the reference point for any variable in a variable constraint related to a location in ρ_1', then ρ satisfies $\mathcal{R}(v, \varphi)$ if and only if ρ' satisfies $\mathcal{R}(v, \varphi)$ where $\rho' = \rho_1 \xrightarrow{(\phi,\psi)} \rho_2^3 \xrightarrow{(\phi',\psi')} \rho_1'$.

Proof. Suppose that $\rho_2^k = \rho_{21} \xrightarrow{(\phi'',\psi'')} \rho_3^{k-2} \xrightarrow{(\phi'',\psi'')} \rho_{21}'$, and

$$\rho_3 = v_1 \xrightarrow{(\phi_1,\psi_1)} v_2 \xrightarrow{(\phi_2,\psi_2)} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n .$$

It follows that $\rho = \rho_1'' \xrightarrow{(\phi'',\psi'')} \rho_3^{k-2} \xrightarrow{(\phi'',\psi'')} \rho_1'''$, and $\rho' = \rho_1'' \xrightarrow{(\phi'',\psi'')} \rho_3 \xrightarrow{(\phi'',\psi'')} \rho_1'''$. The half of the claim, if ρ satisfies $\mathcal{R}(v, \varphi)$ then ρ' satisfies $\mathcal{R}(v, \varphi)$, can be proved as follows. Since ρ satisfies $\mathcal{R}(v, \varphi)$, suppose that the corresponding timed sequence $\sigma = \sigma_1''\hat{}\sigma_r\hat{}\sigma_1'''$ satisfies the condition given in Definition 3.1 where σ_r corresponds to ρ_3^{k-2}. It follows that $\sigma_r = \sigma_{r1}\hat{}\sigma_{r2}\hat{}\ldots\hat{}\sigma_{rk-2}$, and that each σ_{ri} $(1 \leq i \leq k-2)$ is of the form $(v_1, t_1)\hat{}(v_2, t_2)\hat{}\ldots\hat{}(v_n, t_n)$ such that $t_1, t_2, \ldots, t_n$ satisfies the condition in Definition 2.2. Since ρ_2^k is closed in ρ and any location in ρ_1 is not the reference point for any variable in a variable constraint related to a location in ρ_1', by removing $\sigma_{r2}\hat{}\sigma_{r3}\hat{}\ldots\hat{}\sigma_{rk-2}$ from σ we get a timed sequence σ' which satisfies the condition in Definition 3.1 and corresponds ρ'. It follows that ρ' satisfies $\mathcal{R}(v, \varphi)$. The other half of the claim can be proved as follows. Since ρ' satisfies $\mathcal{R}(v, \varphi)$, suppose that the corresponding timed sequence $\sigma' = \sigma_1''\hat{}\sigma_3\hat{}\sigma_1'''$ satisfies the condition given in Definition 3.1 where σ_3 corresponds to ρ_3. Since ρ_2^k is closed in ρ and any location in ρ_1 is not the reference point for any variable in a variable constraint related to a location in ρ_1', by replacing σ_3 with $\underbrace{\sigma_3\hat{}\sigma_3\hat{}\ldots\hat{}\sigma_3}_{k-2}$ in σ' we get a timed sequence σ which satisfies the condition in Definition 3.1 and corresponds ρ. It follows that ρ satisfies $\mathcal{R}(v, \varphi)$. □

Let $H = (X, V, E, v_I, \alpha, \beta)$, and ρ be a path in H of the form

$$v_0 \xrightarrow{(\phi_0,\psi_0)} \ldots \xrightarrow{(\phi_{i-1},\psi_{i-1})} v_i \xrightarrow{(\phi_i,\psi_i)} \ldots \xrightarrow{(\phi_{j-1},\psi_{j-1})} v_j \xrightarrow{(\phi_j,\psi_j)} \ldots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n .$$

A variable constraint $a \leq \sum_{k=0}^{m} c_k x_k \leq b$ related to v_j $(1 \leq j \leq n)$ is *positive* in ρ if the following condition holds:

- $c_k \geq 0$ for any k $(0 \leq k \leq m)$, and
- for any x_k $(0 \leq x \leq m)$, if the reference point is v_i, then any v_l $(i < l \leq j)$ is such that if $\dot{x_k}= [a, b] \in \beta(v_l)$ then $a \geq 0$,

and we say that $b - \sum_{i=0}^{k} c_k d_k$ is the *bound* of the variable constraint where d_k $(0 \leq k \leq m)$ is the reference value of x_k on v_j.

Let $H = (X, V, E, v_I, \alpha, \beta)$ be a linear hybrid automaton, $\mathcal{R}(v, \varphi)$ be a reachability specification, and $\rho = \rho_1 \xrightarrow{(\phi,\psi)} \rho_2^k \xrightarrow{(\phi',\psi')} \rho_1'$ be a path in H where $\rho_2^k = \rho_{21} \xrightarrow{(\phi'',\psi'')} \rho_3^{k-2} \xrightarrow{(\phi'',\psi'')} \rho_{21}'$ $(k > 3)$ is closed in ρ, and

$$\rho_3 = v_1 \xrightarrow{(\phi_1,\psi_1)} v_2 \xrightarrow{(\phi_2,\psi_2)} \dots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n\,.$$

If there is a positive variable constraint $a \leq \sum_{i=0}^{m} c_i x_i \leq b$ related to a location in ρ_1' such that

- there is a variable set $\omega \subseteq \{x_0, x_1, \dots, x_m\}$ $(\omega \neq \emptyset)$ such that for any $x \in \omega$, its reference point is in ρ_1, and
- $\xi > 0$ where ξ is the infimum of the set

$$\left\{ \sum_{i=0}^{m} c_i'\delta_i \;\middle|\; \begin{array}{l} \text{if } x_i \in \omega \text{ then } c_i' = c_i \text{ else } c_i' = 0 \text{ for any } i\ (0 \leq i \leq m); \\ \text{for any } i\ (0 \leq i \leq m),\ \delta_i = u_{i1}t_1 + u_{i2}t_2 + \ldots + u_{in}t_n \\ \text{where } \dot{x_i}= [u_{ij}, u_{ij}'] \in \beta(v_j) \text{ for any } j\ (1 \leq j \leq n); \text{ and} \\ (v_1, t_1)\hat{\ }(v_2, t_2)\hat{\ }\ldots\hat{\ }(v_n, t_n) \text{ is a timed sequence such that} \\ t_1, t_2, \ldots, t_n \text{ satisfy the condition in Definition 2.} \end{array} \right\}$$

(notice that ξ can be calculated by linear programming),

then we say that p_2^k is *constrained* by $\lfloor \zeta/\xi \rfloor + 3$ where ζ is the bound of the variable constraint $a \leq \sum_{i=0}^{m} c_i x_i \leq b$.

Theorem 4.2 Let $H = (X, V, E, v_I, \alpha, \beta)$ be a linear hybrid automaton, $\mathcal{R}(v, \varphi)$ be a reachability specification, and $\rho = \rho_1 \xrightarrow{(\phi,\psi)} \rho_2^k \xrightarrow{(\phi',\psi')} \rho_1'$ be a path in H where ρ_2^k $(k > 3)$ is closed in ρ, and constrained by k'. If $k > k'$ then ρ does not satisfy $\mathcal{R}(v, \varphi)$.

Proof. Suppose that $\rho_2^k = \rho_{21} \xrightarrow{(\phi'',\psi'')} \rho_3^{k-2} \xrightarrow{(\phi'',\psi'')} \rho_{21}'$, and

$$\rho_3 = v_1 \xrightarrow{(\phi_1,\psi_1)} v_2 \xrightarrow{(\phi_2,\psi_2)} \dots \xrightarrow{(\phi_{n-1},\psi_{n-1})} v_n\,.$$

It follows that $\rho = \rho_1'' \xrightarrow{(\phi'',\psi'')} \rho_3^{k-2} \xrightarrow{(\phi'',\psi'')} \rho_1'''$. Suppose that ρ satisfies $\mathcal{R}(v, \varphi)$, and the corresponding timed sequence $\sigma = \sigma_1''\hat{\ }\sigma_r\hat{\ }\sigma_1'''$ satisfies the condition given in Definition 3.1 where σ_r corresponds to ρ_3^{k-2}. It follows that

$$\sigma_r = \sigma_{r1}\hat{\ }\sigma_{r2}\hat{\ }\ldots\hat{\ }\sigma_{rk-2}$$

where each σ_{ri} $(1 \leq i \leq k-2)$ is of the form $(v_1, t_1)\hat{\ }(v_2, t_2)\hat{\ }\ldots\hat{\ }(v_n, t_n)$

such that $t_1, t_2, \dots, t_n$ satisfies the condition in Definition 2.2. Since ρ_2^k is closed in ρ and constrained by k', if $k > k'$ then there is a positive variable constraint related to a location in ρ_1' which is not satisfied, which results in a contradiction and hence, the claim holds. □

This theorem tells us that in some cases we just need to focus a shorter path since extending the path by repeating a path segment in it will result in that the given reachability specification is not satisfied.

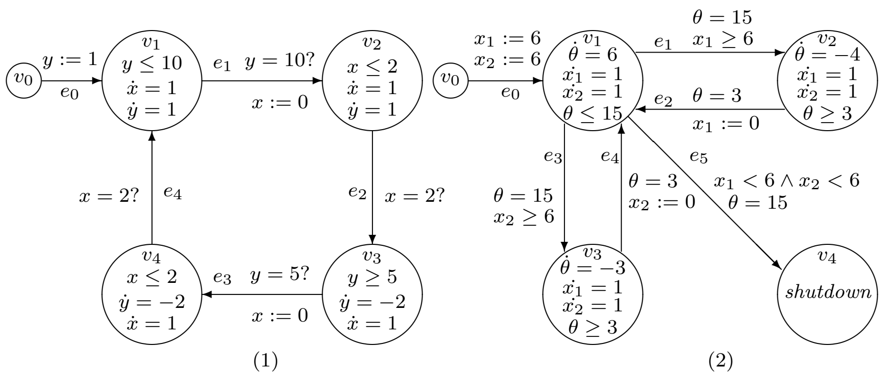


Fig. 1. The automata modelling water-level monitor and temperature control system

5 Tool Prototype and Case Studies

Based on the techniques presented in this paper, we have implemented a tool prototype for the bounded reachability analysis of linear hybrid automata. The tool is implemented in Java, and its graphical interface allows the users to construct, edit, and analyze linear hybrid automata interactively. The linear programming software package which is integrated in the tool is from OR-Objects of DRA Systems [11] which is a free collection of Java classes for developing operations research, scientific and engineering applications. On a HP workstation (Intel Xeon CPU 2.8GHz×2/3.78GB), we evaluated the potential of the techniques presented in this paper by several case studies.

One example depicted in Figure 1(1) is the water-level monitor in [2]. Along the path $v_0\hat{}(v_1\hat{}v_2\hat{}v_3\hat{}v_4)^k$, we check if the location v_4 is reachable, and get the positive answers from the tool with

$$k = 100, 200, 230, 400, 450, 500, 10000$$

respectively. Table 1 shows the tool performance when using the original technique (without any optimization), the optimization technique of decomposing linear programs, and the optimization techniques of shortening paths respectively. When $k \ge 500$, without one of the optimization techniques the tool cannot give a result because of the "Java.lang.out of memory" error occurring in the linear programming package integrated in the tool. Actually, with the optimization technique of shortening paths (see Theorem 4.1), for this example the tool can give a result for any k.

Another example depicted in Figure 1(2) is the temperature control system in [2]. Along the paths

$$v_0\hat{}(v_1\hat{}v_2\hat{}v_1\hat{}v_3)^k\hat{}v_1\hat{}v_4 \quad \text{and} \quad v_0\hat{}(v_1\hat{}v_2)^{k_1}\hat{}(v_1\hat{}v_3)^{k_2}\hat{}v_1\hat{}v_4\,,$$

we check if a complete shutdown is required (the location 4 is reachable), and get the negative answers with the various values of k, k_1, and k_2. The tool performance is shown in Table 2. For the path $v_0\hat{}(v_1\hat{}v_2\hat{}v_1\hat{}v_3)^k\hat{}v_1\hat{}v_4$, no optimization technique works, and the tool cannot give a result when $k \geq 450$ because of the "Java.lang.out of memory" error occurring in the linear programming package integrated in the tool. For the path $v_0\hat{}(v_1\hat{}v_2)^{k_1}\hat{}(v_1\hat{}v_3)^{k_2}\hat{}v_1\hat{}v_4$, the condition of Theorem 4.2 holds so that the optimization technique of shortening paths works.

We also compare our technique with PHAVer [9] which is the improvement of the state-of-the-art tool HYTECH [8]. Because of performing expensive polyhedra computation, the capacity of PHAVer is restricted by the variable number in the automata. We simply construct an experimental automaton depicted in Figure 2 in which there are seven locations and variables. Along the path $v_0\hat{}(v_1\hat{}v_2\hat{}v_3\hat{}v_4)^k\hat{}v_5\hat{}v_6$, we check if the location v_6 is reachable by PHAVer and our tool respectively. Because PHAVer does not provide any timer, we manually record its execution time. The experimental result is shown in Table 3. When k is set to 20 and 30, PHAVer spends about 0.66 and 4 hours respectively for checking, which are much longer than the execution time of our tool with the original technique. PHAVer can not give any result when $k = 40$ after running for 20 hours, but even when $k = 150$ our tool can give the result in a tolerable duration. Notice that for fairness, we use the unfolded path as the input of PHAVer for avoiding it doing the full reachability analysis. Because of performing expensive polyhedra computation, the algorithm complexity of PHAVer is exponential in the number of variables of an automaton, which gives an intuitional explanation for the experiment result.

The above experiments are preliminary and use freely available linear programming packages, but they indicate a clear potential of the techniques presented in this paper with the support of an advanced commercial linear programming package.

6 Conclusion

In this paper, based on linear programming we develop the techniques towards an efficient path-oriented tool for the bounded reachability analysis of linear hybrid automata, which checks one path at a time where the length of the path being checked can be made very large and the size of the automaton can be

Path: $v_0\hat{}(v_1\hat{}v_2\hat{}v_3\hat{}v_4)^k$						
k	Original technique				Decomposing LPs	Shortening paths
	constraints	variables	memory	time	time	time
100	3191	997	512M	61.172s	1.031s	0.031s
200	6391	1997	512M	466.140s	1.562s	0.031s
230	7351	2297	512M	702.766s	1.750s	0.031s
400	12791	3997	1470M	3969.421s	3.187s	0.031s
450	14391	4497	1470M	4485.328s	3.469s	0.031s
500	Java.lang.out of memory error				4.109s	0.031s
10000	Java.lang.out of memory error				38.047s	0.031s

Table 1
Experimental results on the water-level monitor

Path: $v_0\hat{}(v_1\hat{}v_2\hat{}v_1\hat{}v_3)^k\hat{}v_1\hat{}v_4$						
k	Original technique				Decomposing LPs	Shortening paths
	constraints	variables	memory	time	time	time
100	4415	1004	512M	90.218s	90.218s	90.218s
200	8815	2004	512M	686.938s	686.938s	686.938s
230	10315	2304	512M	1180.297s	1180.297s	1180.297s
400	17591	3998	1470M	5574.312s	5574.312s	5574.312s
450	Java.lang.out of memory error				Java.lang.out of memory error	

Path: $v_0\hat{}(v_1\hat{}v_2)^{k_1}\hat{}(v_1\hat{}v_3)^{k_2}\hat{}v_1\hat{}v_4$							
k_1	k_2	Original				Decomposing LPs	Shortening paths
		constraints	variables	memory	time	time	time
50	50	2215	504	512M	10.532s	10.532s	0.016s
100	100	4415	1004	512M	76.703s	76.703s	0.016s
200	200	8791	1998	1004M	496.609s	496.609s	0.016s

Table 2
Experimental results on the temperature control system

Path: $v_0\hat{}(v_1\hat{}v_2\hat{}v_1\hat{}v_3)^k\hat{}v_5\hat{}v_6$		
k	PHAVer	Our tool (original technique)
	time	time
20	$\approx$ 2400s	12.359s
30	$\approx$ 4h	36.688s
40	no result after 20 hours	82.891s
80		616.671s
100		1143.344s
150		4067.391s

Table 3
Experimental results on the experimental automaton

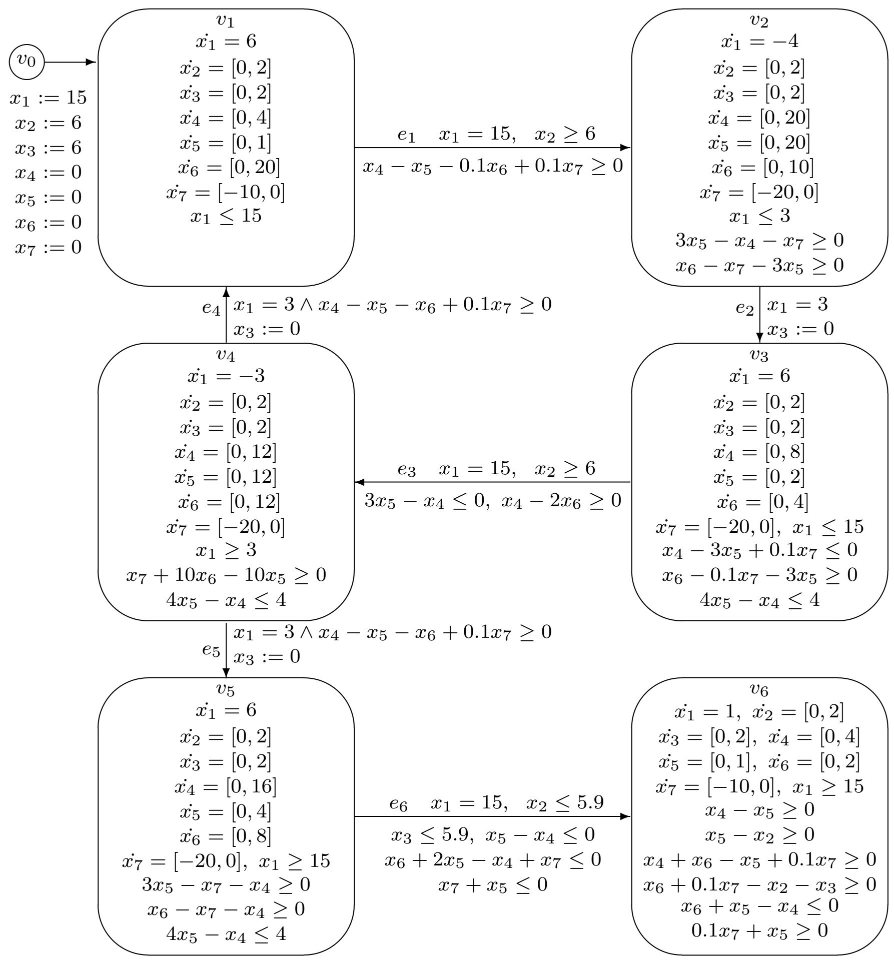


Fig. 2. An experimental automaton

made large enough to handle problems of practical interest. Since the existing techniques have not resulted in an efficient tool for checking all the paths in a linear hybrid automaton for problems with sizes of practical interest, the tool derived from the techniques presented in this paper will become a design engineer's assistant for the reachability analysis of linear hybrid automata.

References

[1] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pp. 278-292.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H.Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine. The algorithmic analysis of hybrid systems. In *Theoretical Computer Science*, 138(1995), pp.3-34.

[3] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's Decidable About Hybrid Automata? In *Journal of Computer and System Sciences*, 57:94-124, 1998.

[4] Thomas A. Henzinger, Pei-Hsin Ho, Howard Wong-Toi. Algorithmic Analysis of Nonlinear Hybrid Systems.In *IEEE Transactions on Automatic Control*, 1998, pp.540-554.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, Yunshan Zhu. Bounded Model Checking. In *Advance in Computers*, Vol.58, Academic Press, 2003.

[6] Martin Franzle, Christian Herde. Eifficient Proof Engines for Bounded Model Checking of Hybrid Systems. In *Electronic Notes in Theoretical Computer Science*, Vol.89, No.4, 2004.

[7] Gilles Audemard, Marco Bozzano, Alessandro Cimatti, Roberto Sebastiani. Verifying Industrial Hybrid Systems with MathSAT. In *Electronic Notes in Theoretical Computer Science*, Vol.89, No.4, 2004.

[8] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: a model checker for hybrid systems. In *Software Tools for Technology Transfer*, 1:110-122, 1997.

[9] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems past HyTech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control (HSCC'05)*, Lecture Notes in Computer Science 2289, Springer-Verlag, 2005, pp.258-273.

[10] Li Xuandong, Zhao Jianhua, Pei Yu, Li Yong, Zheng Tao, and Zheng Guoliang. Positive Loop-Closed Automata: A Decidable Class of Hybrid Systems. In *Journal of Logic and Algebraic Programming*, Vol.52-53(C), Elsevier Science, 2002, pp.79-108.

[11] OR-Objects of DRA Systems.
http://OpsResearch.com/ OR-Objects/index.html.

Acknowledgement

We would like to thank Prof. Edmund M. Clarke and the anonymous reviewers for their valuable comments and suggestions. The first and third authors are supported by the National Natural Science Foundation of China (No.60425204, No.60233020), the National Grand Fundamental Research 973 Program of China (No.2002CB312001), and by the Jiangsu Province Research Foundation (No.BK2004080). The second author acknowledges the support of a graduate fellowship from the Computer Science Department at Carnegie Mellon.

Compressing BMC Encodings with QBF

Toni Jussila [1] Armin Biere [2]

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract

Symbolic model checking is PSPACE complete. Since QBF is the standard PSPACE complete problem, it is most natural to encode symbolic model checking problems as QBF formulas and then use QBF decision procedures to solve them. We discuss alternative encodings for unbounded and bounded safety checking into SAT and QBF. One contribution is a linear encoding of simple path constraints, which usually are necessary to make k-induction complete. Our experimental results show that indeed a large reduction in the size of the generated formulas can be obtained. However, current QBF solvers seem not to be able to take advantage of these compact formulations. Despite these mostly negative results the availability of these benchmarks will help to improve the state-of-the-art of QBF solvers and make QBF based symbolic model checking a viable alternative.

Key words: Bounded Model Checking, Encoding, QBF, SAT.

1 Introduction

Bounded Model Checking (BMC) [3] has the motivation to improve on BDD based symbolic model checking by using SAT procedures. Already in the original paper the use of QBF decisions procedures was suggested as a tool to make BMC complete without using BDDs. Completeness means that an LTL property can also be shown to hold as opposed to just being able to find counter examples. In this paper we focus on simple safety properties, for which we want to prove that a *bad* state is not reachable. More general properties can be handled for instance through techniques from [14].

The completeness result of [3] uses the fact that the diameter of the system, which is the length of the longest shortest path between two states, is an upper bound on the length of potential counter examples. The question is how the diameter can be calculated. In [3] a QBF formula is presented parameterized

[1] Email: `toni.jussila@jku.at`
[2] Email: `armin.biere@jku.at`

by d, which is satisfiable resp. *true*, iff d is an upper bound on the diameter. Nevertheless this formulation was not used in the experiments, since efficient QBF solvers did not exist at that time.

In graph theory the notion of diameter is also known as *eccentricity*. To determine the eccentricity of the state transition graph of sequential circuits has been investigated in [10]. The authors used a dedicated QBF solver for quantifier elimination. However, the examples that could be handled are tiny. An argument why DPLL style QBF solvers can not handle this kind of problems well is given in [17]: in essence DPLL style QBF solvers need to perform an explicit state space search to determine the diameter.

However, it is possible to generate a purely propositional SAT formula without quantifiers for almost the same problem [3]. If it is unsatisfiable, it constitutes an upper bound on the diameter. This formula is parameterized by r and is unsatisfiable iff there is no *cycle free* path of length r. In graph terminology a cycle free path is also called *simple* path, while in [3] the maximal length of a simple path is called *reoccurrence diameter*.

These concepts can be refined in two ways [15]: first the diameter and the reoccurrence diameter can be *initialized*. The paths, both in the QBF and in the SAT case, can be forced to fulfill the additional constraint that exactly the first state is an initial state. Furthermore, instead of looking for maximal simple paths starting from an initial state in a *forward* manner, one can work *backward* from a bad state. In particular the maximal length of a simple path for which exactly its last state is a bad state is also an upper bound on the maximal length of counter examples that need to be searched. We call such paths *terminal*.

In the special case $k = 1$ this technique amounts to check that the *good* states are an inductive invariant of the transition relation. Therefore the technique is also known as k-induction [15]. It seems to be much more successful in practice than forward checking, since it can utilize locality of properties, even if it is just implicitly through the SAT solver, while a forward formulation will need to take all state bits into account.

However, simple paths can be exponentially larger than their corresponding diameters, both in forward and backward reasonings. Therefore the question still remains, whether an approach using QBF reasoning would not allow to terminate the search for counter examples much earlier. Also the state-of-the-art in QBF solver technology improved considerably in recent years [11].

To our knowledge, there are no published results on using QBF for backward reasoning yet. Unfortunately our experimental results for backward reasoning provide a strong indication that similar to the forward reasoning results of [10,17] QBF based fixpoint algorithms can not yet really compete with BDD based or other complete model checking algorithms using SAT as discussed for instance in [1]. Not a single instance was solved that could not be solved with k-induction as well.

On the positive side we provide new compact formulations of bounded model checking problems and also show that in certain cases QBF based reasoning can outperform SAT based reasoning. Our benchmarks will be made publicly available. They will help to improve the state-of-the-art of QBF solvers and hopefully lead to efficient QBF based model checking algorithms.

Finally, we experimented with functional and relational unrollings of the next state logic. The experiments clearly show, that a functional unrolling is much more compact. The generated CNF is much smaller when using syntactic substitution for next state functions instead of conjoining the transition relations. The run times of the SAT solver also decreases considerably.

2 Background

Quantified boolean formulas (QBF) form a propositional logic with quantifiers over boolean variables. The QBF solvers we use only accept QBF in conjunctive normal form (CNF) in prenex form. The standard algorithm [18] for producing CNF for SAT can also be used for QBF after pulling out the quantifiers. The additional variables will be existentially quantified in the innermost scope. In the rest of the paper we do not require prenex CNF.

Our system model is the standard relational model used in symbolic model checking. It is a flat boolean encoded Kripke structure K with initial state constraint $I(s)$, transition relation $T(s, s')$, bad state constraint $B(s)$ and good state constraints $G(s)$ with $G(s) \equiv \neg B(s)$. Evaluations $\sigma \in 2^n$ of state variable vectors s, s', $\dots$ act as states. A state variable vector, in the following just short *state variable*, is made up of n individual *state bits*, which are just boolean variables. Individual state bits and equalities over state variables serve as atomic propositions.

A valid *path* of length k in K is an evaluation of state variables $s_0, \dots s_k$ that satisfies the *path constraint* $T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k)$. An *initialized path constraint* requires in addition that $I(s_0)$ holds, while a *terminal path constraint* requires that $B(s_k)$ holds. A bad state is reachable iff there is a satisfiable path constraint for some k, which at the same time is initial and terminal. In this paper we only consider the problem of checking, whether a bad state is reachable.

3 Fixpoints

The algorithm of Fig. 1 is the standard BFS algorithm for checking simple safety properties with BDDs. The sets C and N as well as the relations I, T, and B are represented symbolically. The algorithm implements a fixpoint computation starting from the initial states, adding the next states N reachable in one step, with $Img(C)(s') \equiv \exists s[T(s, s') \wedge C(s)]$, from the *current* states reached so far C until either a bad state B is found or the loop terminates. The focus of BMC is the former while in this paper we concentrate on checking the loop condition.

```
model-check$^\mu_{\text{forward}}$ $(I,\ T,\ B)$
  $C = \mathit{false}$; $N = I$;
  while $N \not\Leftrightarrow C$ do
    if $B \wedge N$ satisfiable then
      return "bad state reachable";
    $C = N$;
    $N = C \vee \mathit{Img}(C)$;
  done;
  return "no bad state reachable";
```

Fig. 1. On-the-fly forward model checking algorithm for safety properties.

The loop condition is invalid initially, and the loop is not even entered, iff $I = \mathit{false}$, or equivalently if $I(s)$ is unsatisfiable. This can be checked by a SAT solver. The validity of the loop condition, after the first iteration can also be checked by a SAT solver, since it is equivalent to the satisfiability of $\exists s, s'[I(s) \wedge T(s, s') \wedge \neg I(s')]$. If this formula is unsatisfiable then I actually turns out to be an inductive invariant of the transition relation.

However, after the second iteration the loop condition is equivalent to the satisfiability of

$$\begin{aligned}\exists s_0, s_1, s_2[\ & I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \\ & \forall t_0, t_1[I(t_0) \wedge T(t_0, t_1) \rightarrow (s_2 \neq t_0 \wedge s_2 \neq t_1)]]\end{aligned}$$

which is a proper QBF formula with one alternation.[3] In general, the loop condition is fulfilled after k iterations iff the following formula is satisfiable:

$$\begin{aligned}\exists s_0, s_1, \ldots, s_k[\ & I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \\ & \forall t_0, t_1, \ldots, t_{k-1}[\ I(t_0) \wedge T(t_0, t_1) \wedge \ldots \wedge T(t_{k-2}, t_{k-1}) \rightarrow \\ & \qquad (s_k \neq t_0 \wedge \ldots \wedge s_k \neq t_{k-1})]]\end{aligned}$$

Variations of this formulation, were also used in [10,17]. Their practical usage is rather restricted. There was not a single instance in our experiments, where initialized diameter checking, was doable this way, if it involved any alternation of quantifiers. Clearly much stronger QBF solvers are required.

Initial experiments in using the reoccurrence diameter were also unsuccessful. The instances are solvable for small k, but the reoccurrence diameter turns out to be too large for these examples and the SAT instances also become intractable very soon. This is in contrast to the experience with simple path constraints in k-induction. Therefore we suggest to represent the termination check for symbolic backward fixpoint computation as QBF decision

[3] Here we need the common assumption that the transition relation T is total, which we will assume for the rest of the paper.

problem as well:

$$\begin{aligned}\exists s_0, s_1, \ldots, s_k[\ & T(s_0,s_1) \wedge \ldots \wedge T(s_{k-1},s_k) \wedge B(s_k) \wedge \\ & \quad \forall t_0, t_1, \ldots, t_{k-1}[\ T(t_0,t_1) \wedge \ldots \wedge T(t_{k-2},t_{k-1}) \wedge B(t_{k-1}) \rightarrow \\ & \qquad\qquad (s_0 \neq t_0 \wedge \ldots \wedge s_0 \neq t_{k-1})]]\end{aligned}$$

In our experiments it turns out that in this case two instances for $k = 2$ could be solved, for which also k-induction determined termination easily. Nevertheless, this negative result still shows, that even when using QBF, backward computation may be superior to forward computation as it is the case with checking reoccurrence diameters versus k-induction.

For backward fixpoint calculations we actually used a slightly different formulation, as also used in SAT based k-induction [15], where T is replaced by T_G with $T_G(s,s') \equiv G(s) \wedge T(s,s')$. If the formula is unsatisfiable then k is a bound on the maximum length of paths that have to be searched in order find a path to a bad state, which only traverses good states except for the last state. This optimization may reduce the bounds that have to be checked considerably.

4 Non-Copying Iterative Squaring

Following the classical proof of PSPACE hardness of QBF [13,16] we can use *non-copying iterative squaring* to compute symbolically the transitive closure of the transition relation as follows:

$$\begin{aligned}T^{2\cdot i}(s,s') \ \equiv\ \forall c\,[\,\exists\, l,m,r\,[\,(c \rightarrow (\ \ l = s\ \ \wedge r = m))\ \wedge \\ (\bar{c} \rightarrow (\ \ l = m \wedge r = s'))\ \ \wedge\ T^i(l,r)]]\end{aligned}$$

The universal "choice variable" c just instantiates the formal parameters (l,r) of $T^i(l,r)$ with either the actual parameters (s,m) in the positive case or (m,s') in the negative case. This is simply a compact QBF reformulation of *copying iterative squaring*

$$T^{2\cdot i}(s,s') \ \equiv\ \exists\, m\,[\,T^i(s,m) \wedge T^i(m,s')]$$

which doubles the size of the formula with every application, while the non-copying formulation just adds some state variable equalities each time.

A similar formulation was discussed in [12] and has also been used in [2] to perform bounded model checking of very simple counter circuits. In the latter paper it has been observed that current state-of-the-art QBF solvers can barely keep up with SAT based bounded model checking on these examples. However, the QBF formula is linear in the model and logarithmic in the number of steps, which gives an at most quadratic formula in the number of state bits. The worst case only occurs if the sequential depth, e.g. the initialized diameter, really turns out to be 2^n.

5 Simple Path Constraints

In [3,15] the concept of *simple path constraints* was introduced

(1) $$\bigwedge_{0\leq i<j\leq k} s_i \neq s_j$$

Note that the size of this formula is quadratic in k and each $s_i \neq s_j$ involves the comparison of n state bits. By sorting the s_i symbolically as in [9] an $\mathcal{O}(k \cdot log(k))$ size bound can be obtained. In practice, due to large constants, simpler sorting networks with size $\mathcal{O}(k \cdot (log(k))^2)$ are preferred, such as *bitonic sort* or *odd-even mergesort*. In our experiments we used the latter, since it requires slightly less comparisons than the bitonic sorting network used in [9].

If these constraints are conjoined with path constraints, they allow to obtain a complete model checking procedure. If the path constraints are initialized and the result becomes unsatisfiable, then k is a bound on the reoccurrence diameter [3]. If no counter example up to this length exists, no bad state is reachable. Similarly, if the simple path constraints are conjoined with a terminal path constraint, as in k-induction [15], then the unsatisfiability of the result, again shows that k is an upper bound on the maximal length of counter examples that need to be considered. The formula that is checked in k-induction is the following:

(2) $$\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \ \wedge \ \bigwedge_{i=0}^{k-1} G(s_i) \ \wedge \ B(s_k) \ \wedge \ \bigwedge_{0\leq i<j<k} s_i \neq s_j$$

Note, that the last state s_k as a "good state" can never be equal to one of the previous "bad states". Therefore, from Eqn. 1 we can remove comparisons with the last state in k-induction. A similar argument could be used for computing the reoccurrence diameter.

5.1 Compact Simple Path Constraints in QBF

One of our contribution of this paper is a reformulation of the simple path constraints of Eqn. 1 in QBF as follows:

(3) $$\forall l_0, \ldots, l_k \ [\ \exists s \ [\ |\sum_{i=0}^{k} l_i| = 1 \ \rightarrow \ \bigwedge_{i=0}^{k} (l_i \ \leftrightarrow \ (s = s_i))]$$

The resulting formula needs one alternation of quantifiers and is linear in k as opposed to quadratic complexity of the original formula. Note that the state variables of the corresponding path constraints are free variables of this formula and are quantified existentially in the outermost scope for our applications.

In order to obtain linear complexity the cardinality constraint $|\Sigma_{i=0}^{k} l_i|$, which simply states that exactly one of the l_i is true, has to be encoded with a linear sized circuit. This is easily possible, since for instance the ROBDD for this cardinality constraint for any variable order has linear size in k.

The additional "bits" $l_0, \ldots, l_k$ provide a one-hot encoding of the index of a reference vector, which is saved in s and is enforced to be different from all the other state vectors, e.g. l_i saves s_i as s and forces s to be different from all other s_j with $i \neq j$. A binary encoding of the index of the reference vector is also possible and requires only $\lceil log_2 k \rceil$ additional universal variables.

This example already shows some of the modelling power of QBF, which, we believe, is hardly used in practice yet. But we can go one step further by sharing the transition relation across time frames as in [6]. Our QBF reformulation following [6] of path constraints is as follows:

$$\forall l_0, \ldots, l_{k-1} \; [\; \exists s, s' \; [T(s, s') \wedge \bigwedge_{i=0}^{k-1} (l_i \; \rightarrow \; (s = s_i \wedge s' = s_{i+1}))] \tag{4}$$

Putting both together we obtain a compact reformulation of simple path constraints in QBF with transition relation sharing:

$$\begin{aligned} &\forall l_0, \ldots, l_k \; [\; \exists s, s' \; [\; T(s, s') \wedge \\ &\qquad \textstyle\bigwedge_{i=0}^{k-1} (l_i \; \rightarrow \; (s = s_i \wedge s' = s_{i+1})) \wedge \\ &\qquad (|\textstyle\sum_{i=0}^{k} l_i| = 1 \; \rightarrow \; \bigwedge_{i=0}^{k} (l_i \; \leftrightarrow \; (s = s_i)))] \end{aligned} \tag{5}$$

Initial respectively terminal constraints can be added as needed. In the context of k-induction practical experience shows that adding good state constraints to the current state of the transition relation, as in Eqn. 2, improves performance and particularly decreases the bound k considerably. We can achieve the same effect in the QBF formulation by just adding $G(s)$ to the innermost existential scope, thus, actually sharing G across time frames as well. In the experiments we used the latter version.

6 Transition Functions and Relations

SMV [5] allows two ways to specify the transitions that a system can make. We refer to these as functional and relational part. In the functional part (inside an `ASSIGN` section of the SMV file) the value of a variable in the next state is defined as a boolean function of the variable values in the current state. The relational part is simply a boolean formula where the atomic formulas are current and next state variables and this formula (given in the `TRANS` section of the SMV file) has to hold between any two states. An SMV file can contain both a relational and functional part to describe the system's transition relation.

A functional transition relation allows an optimization in the translation to SAT/QBF when the transition relation (or the simple path constraint) is unrolled. Namely, for a functional state variable, it is possible to substitute its next state function in any state after the initial state. Thereafter, the representation can be simplified by propagating information from the initial state to subsequent states. Consider for instance variables x_0 and x_1 and let the SMV file contain the definitions `init(x0) := 0` and `next(x1) := !x0`. Then it is obviously possible to infer that the value of x_1 after the transition relation is unrolled once is 1 etc. We refer to this optimization as *functional substitution*. Notice that this substitution is not possible if QBFs are used to share the transition relation and the simple state constraint. In our experiments, we compare this optimized translation to a translation where functional substitution is disabled (considering the model be purely relational). Our experimental results show that in some cases the optimization that functionality allows plays an important role.

7 Experiments

We have implemented our approach in a tool called SMV2QBF. It reads flat SMV specifications with simple safety properties as input and translates them to QBFs. The tool has several switches corresponding to different model checking problems. It is possible to perform standard BMC, compute diameter and reoccurrence diameter, compute fixpoint, and do k-induction proofs [8]. For most of these switches, there are two or more encodings, the standard propositional one and one using more compact QBFs.

We present two sets of results, first of problems where it is possible to prove that the safety property holds using k-induction (the induction step eventually becomes unsatisfiable). Second, we have examples where a counterexample is found using standard BMC.

For the experiments we used a cluster of Pentium IV 3.0 GHz PCs with 2GB of main memory running Debian Sarge Linux. The time limit was set to 1000 seconds and the memory limit to 1GB of main memory. The examples that we use are from the TIP tool by Eén and Sörensson [8]. We use QUANTOR (version 2.13) [2] as the QBF solver. QUANTOR uses a SAT solver as a back end and for this purpose we use PICOSAT (version 1.251). We also compare QUANTOR to another state-of-the-art QBF solver, QUBE (version 1.3) [7]. For every instance we tried, QUANTOR performed better.

The results for the examples where the property is proven are shown in Tables 1 and 2. The columns of the tables are as follows. In both tables, the two leftmost columns give the name of the example and the bound needed to prove the property. Thereafter are 6 columns of the form s(x) (Table 1) and t(x) (Table 2), where x is the type of encoding, s(x) stands for size and t(x) for time. We use the following encodings for k-induction step:

(i) i is the standard fully propositional encoding (Eqn. 2),

(ii) ir is i without functional substitution (see Sect. 6),

(iii) is implements simple state constraints with sorting networks,

(iv) isr is is without functional substitution,

(v) l is an encoding where the transition relation is unrolled but the simple path constraints are encoded as given in Eqn. 3,

(vi) *lr* is *l* without functional substitution,

(vii) *L* is an encoding using Eqn. 5 using a one-hot index encoding, and

(viii) *B* is a modification of Eqn. 5 with binary index encoding.

A column of the form s(x) gives the size in kilobytes of the (SAT/QBF) formula for encoding x and the bound given in column k. The running time given in column t(x) is the time required to solve the *single* instance of encoding x corresponding to the depth given in the column k. If the entry is of the form N/A then the memory limit was exceeded (we experienced no time outs).

Tables 3 and 4 follow the same conventions as Tables 1 and 2, however, this time k is the smallest depth needed to find a counterexample. The encodings are as follows:

(i) *b*, the standard propositional BMC encoding,

(ii) *br* is *b* without functional substitution,

(iii) *C*, a compact BMC encoding with a single copy of the transition relation (see Eqn. 4), and

(iv) *S*, a BMC encoding with noncopying iterative squaring (see Sect. 4).

Tables 1–4 seem to warrant the following conclusions. Applying QBF representations yields in many cases smaller formulas and the difference seems to grow with larger bounds. This is especially so when the model is fully relational. This conclusion is rather obvious, though, since the QBF grows linearly and in the propositional case a new copy of the transition relation is needed when the bound is incremented.

A perhaps more interesting observation is that the running times of the optimized version of the propositional encodings (columns t(i) and t(b)) are always lower than the encodings using more compact formulas. We identify two reasons for this. First, propositional encoding allows one to perform optimizations (preprocessing steps), like functional substitution (see Sect. 6) but also bounded cone of influence [4] in an efficient manner. [4] Indeed, it should be noted that for some test cases (like the examples vis.prodcell.*), when the SMV model is made fully relational and thus no functional substitutions are possible, QBF encodings sharing the transition relation (columns t(L) and t(B)) perform better than the SAT encoding (column t(ir)).

Second, the research community has invested much more effort to implement efficient SAT solvers than is the case for QBFs. We expect more efficient QBF solvers in the future.

[4] Notice that we always reduce the model by cone of influence reduction in every encoding, particularly for the transition relation and comparing state variables. In addition, we only compare state variables that occur in both current and next states [8].

name	k	s(i)	s(ir)	s(is)	s(isr)	s(l)	s(lr)	s(L)	s(B)
cmu.periodic.N	96	41372	41416	27996	28096	9832	9844	2100	2112
eijk.S208.S	258	146728	170828	49772	72808	3232	26864	3656	3696
eijk.S208c.S	258	148840	186168	51332	82716	3212	33792	3884	3924
eijk.S208o.S	258	129788	164740	44240	77480	3048	37516	2920	2956
eijk.S298.S	58	13868	19752	10740	16584	1136	6812	1668	1672
eijk.S510.S	10	656	2504	1268	3068	372	2404	632	636
eijk.S820.S	11	844	3468	1300	3904	584	3500	664	664
eijk.S832.S	11	900	3592	1400	4072	628	3704	700	700
eijk.S953.S	7	448	2612	748	2912	360	2748	776	776
ken.oop1.C	29	3204	4204	3536	4504	1116	2184	608	608
nusmv.guid*1.C	10	1360	2564	2192	3320	1112	2224	752	752
nusmv.guid*7.C	27	8208	11996	9520	13316	2712	6172	1728	1732
nusmv.tcas2.B	6	1176	3936	1820	4584	1016	4364	1284	1288
nusmv.tcas3.B	5	892	2964	1420	3704	836	3640	1172	1172
texas.par*2.E	2	36	480	44	488	44	548	356	356
vis.prodc*12.E	29	10612	65488	11352	66272	6008	68328	3320	3320
vis.prodc*13.E	8	1556	16112	1884	16488	1472	18792	2444	2444
vis.prodc*14.E	16	4112	33700	4776	34440	3164	37312	2776	2776
vis.prodc*15.E	23	7260	49808	8496	51024	4664	53496	3068	3068
vis.prodc*16.E	5	800	9752	1004	9980	824	11740	2320	2320
vis.prodc*17.E	27	9444	60076	10392	61092	5528	59872	3236	3236
vis.prodc*18.E	13	3036	26284	3708	27044	2500	28616	2652	2652
vis.prodc*19.E	22	6772	47484	8012	48712	4468	51204	3028	3028
vis.prodc*24.E	37	15832	87760	16888	88900	7924	89092	3652	3656

Table 1
k-induction sizes

8 Conclusion

This paper on one hand again provides negative results on using QBF for unbounded model checking and less negative for bounded model checking. On the other hand we were able to show that in practice QBF formulations

name	k	t(i)	t(ir)	t(is)	t(isr)	t(l)	t(lr)	t(L)	t(B)
cmu.periodic.N	96	144.7	144.6	178.4	173.3	162.5	162.3	345.1	153.5
eijk.S208.S	258	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
eijk.S208c.S	258	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
eijk.S208o.S	258	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
eijk.S298.S	58	66.0	61.5	N/A	N/A	N/A	207.7	195.1	132.2
eijk.S510.S	10	1.9	2.4	163.7	174.8	N/A	5.4	5.7	5.9
eijk.S820.S	11	2.2	3.4	93.8	82.1	N/A	5.3	3.6	3.4
eijk.S832.S	11	2.3	3.5	95.5	84.9	N/A	6.0	3.9	3.6
eijk.S953.S	7	0.8	1.7	8.6	8.3	38.4	2.8	3.1	2.8
ken.oop1.C	29	23.0	18.3	N/A	N/A	30.4	30.9	50.3	37.8
nusmv.guid*1.C	10	1.4	2.0	1.5	2.1	6.3	6.2	7.6	6.2
nusmv.guid*7.C	27	61.7	60.2	173.7	85.8	87.5	92.6	116.5	116.7
nusmv.tcas2.B	6	1.2	2.5	1.3	2.6	3.9	4.9	11.1	6.4
nusmv.tcas3.B	5	0.5	1.4	1.0	2.0	3.2	3.4	6.5	4.9
texas.par*2.E	2	0.0	0.2	0.0	0.2	0.1	0.2	0.4	0.2
vis.prodc*12.E	29	30.3	197.9	25.0	173.1	N/A	245.8	74.3	57.0
vis.prodc*13.E	8	1.4	13.3	1.6	12.6	5.0	16.8	8.6	8.1
vis.prodc*14.E	16	5.6	46.6	4.6	44.4	23.7	64.6	20.2	17.6
vis.prodc*15.E	23	15.3	113.5	17.3	100.5	85.0	146.2	43.8	34.4
vis.prodc*16.E	5	0.8	6.2	0.9	6.2	2.1	8.6	7.4	6.9
vis.prodc*17.E	27	32.1	163.7	25.2	145.6	N/A	200.2	64.6	52.1
vis.prodc*18.E	13	3.6	28.7	3.7	29.6	13.9	37.9	14.5	12.7
vis.prodc*19.E	22	12.6	97.6	12.3	94.4	70.6	130.1	36.7	32.0
vis.prodc*24.E	37	59.8	N/A	49.4	N/A	N/A	N/A	125.0	103.9

Table 2
k-induction running times

can be much more compact than SAT instances and sometimes solved faster for relational encodings. Our results clearly show that much more research in QBF is needed to be able to use QBF as alternative to SAT based model checking, even in the bounded case.

The tool SMV2QBF and the benchmarks in DIMACS format are available at http://fmv.jku.at/smv2qbf. We are currently working on producing structural

name	k	s(b)	s(br)	s(C)	s(S)
nusmv.tcas1.B	10	960	5580	4512	1524
nusmv.tcas4.B	14	1604	9256	6496	1516
nusmv.tcas5.B	23	2688	13104	9796	1668
nusmv.tcas6.B	16	2816	14900	10188	1668
texas.parsesys1.E	9	140	2392	2140	568
texas.parsesys3.E	8	100	2088	1812	516
texas.twoproc2.E	15	48	13832	9676	1636
texas.twoproc4.E	19	224	19004	12408	1676
vis.eisenberg.E	19	632	19644	12172	1580

Table 3
BMC sizes

name	k	t(b)	t(br)	t(C)	t(S)
nusmv.tcas1.B	10	0.9	42.3	364.6	56.7
nusmv.tcas4.B	14	1.6	46.8	558.0	56.7
nusmv.tcas5.B	23	3.3	51.4	N/A	81.9
nusmv.tcas6.B	16	3.2	46.9	N/A	76.1
texas.parsesys1.E	9	0.1	10.2	86.4	10.8
texas.parsesys3.E	8	0.1	9.1	69.0	8.8
texas.twoproc2.E	15	0.0	133.4	N/A	141.8
texas.twoproc4.E	19	0.3	147.4	N/A	213.3
vis.eisenberg.E	19	1.1	80.9	N/A	117.5

Table 4
BMC running times

benchmarks as well, in the form of *and-inverter-graphs* (AIGs).

References

[1] Amla, N., X. Du, A. Kuehlmann, R. P. Kurshan and K. L. McMillan, *An analysis of SAT-based model checking techniques in an industrial environment*, in: *Proc. CHARME'05*, LNCS **3725**.

[2] Biere, A., *Resolve and expand*, in: *Proc. SAT'04*, LNCS **3542**.

[3] Biere, A., A. Cimatti, E. Clarke and Y. Zhu, *Symbolic model checking without*

BDDs, in: *Proc. TACAS'99*, LNCS **1579**.

[4] Biere, A., E. Clarke, R. Raimi and Y. Zhu, *Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs*, in: *Proc. CAV'99*, LNCS **1633**.

[5] Cimatti, A., E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, *NuSMV 2: An opensource tool for symbolic model checking*, in: *Proc. CAV'02*, LNCS **2404**.

[6] Dershowitz, N., Z. Hanna and J. Katz, *Bounded model checking with QBF*, in: *Proc. SAT'05*, LNCS **3569**.

[7] E. Giunchiglia, M. Narizzano, A. T., *System description: QuBE A system for deciding quantified boolean formulas satisfiability*, in: *Proc IJCAR'01*, LNCS **2083**.

[8] Eén, N. and N. Sörensson, *Temporal induction by incremental SAT solving*, in: *Proc. BMC'03*, ENTCS **89**.

[9] Kröning, D. and O. Strichman, *Efficient computation of recurrence diameters*, in: *Proc. VMCAI'03*, LNCS **2575**.

[10] Mneimneh, M. and K. Sakallah, *Computing vertex eccentricity in exponentially large graphs: QBF formulation and solution*, in: *Proc. SAT'03*, LNCS **2919**.

[11] Narizzano, M., L. Pulina and A. Tacchella, *Report of the third QBF solvers evaluation*, Journal on Satisfiability, Boolean Modeling and Computation **2** (2006).

[12] Rintanen, J., *Partial implicit unfolding in the Davis-Putnam procedure for quantified boolean formulae*, in: *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'01)*, 2001.

[13] Savitch, W. J., *Relation between nondeterministic and deterministic tape complexity*, Journal of Computer and System Sciences **4** (1970).

[14] Schuppan, V. and A. Biere, *Efficient reduction of finite state model checking to reachability analysis*, Software Tools for Technology Transfer (STTT) **5** (2004), pp. 185–204.

[15] Sheeran, M., S. Singh and G. Stålmarck, *Checking safety properties using induction and a SAT-solver*, in: *Proc. FMCAD'00*, LNCS **1954**.

[16] Stockmeyer, L. J. and A. R. Meyer, *Word problems requiring exponential time*, in: *5th Annual ACM Symposium on the Theory of Computing*, 1973.

[17] Tang, D., Y. Yu, D. Ranjan and S. Malik, *Analysis of search based algorithms for satisfiability of quantified boolean formulas arising from circuit state space diameter problems*, in: *Proc. SAT'04*, LNCS **3542**.

[18] Tseitin, G. S., *On the Complexity of Derivation in Propositional Calculus*, in: *Studies in Constructive Mathematics and Mathematical Logic, Part II*, Seminars in Mathematics **8** (1968).

Invited Talk

Techniques for proving properties with SAT-based MC.

Fabio Somenzi
University of Colorado Dept. of Electrical and Computer Engineering, Boulder, Colorado, USA
Fabio@Colorado.edu

Bounded Model Checking with Parametric Data Structures [1]

Erika Ábrahám Marc Herbstritt Bernd Becker

Albert-Ludwigs-University, Freiburg im Breisgau, Germany

Martin Steffen

Christian-Albrechts-University, Kiel, Germany

Abstract

Bounded Model Checking (BMC) is a successful refutation method to detect errors in not only circuits and other binary systems but also in systems with more complex domains like timed automata or linear hybrid automata. Counterexamples of a fixed length are described by formulas in a decidable logic, and checked for satisfiability by a suitable solver.

In an earlier paper we analyzed how BMC of linear hybrid automata can be accelerated already by appropriate encoding of counterexamples as formulas and by selective conflict learning. In this paper we introduce parametric datatypes for the internal solver structure that, taking advantage of the symmetry of BMC problems, remarkably reduce the memory requirements of the solver.

Key words: BMC, Hybrid Automata, Parametric Data Structures, SAT.

1 Introduction

Bounded model checking (BMC) [10] is a successful, relatively young refutation method which was studied and applied very intensively in the last years, see for example [12,13] for some industrial applications. Starting with the initial states of a system, the BMC algorithm considers computations with increasing length $k = 0, 1, \ldots$. For each k, the algorithm checks whether there exists a *counterexample* of the given length, i.e., if there is a computation that starts in an initial state and that leads to a state violating the system specification in k steps.

[1] This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

Basically, BMC can be applied to all kinds of systems for which reachability within a bounded number of steps can be expressed in a decidable logic. For example, for *discrete* systems first-order predicate logic is used, whereas the analysis of *linear hybrid automata* [4,24] requires first-order logic formulas over $(\mathbb{R}, +, <, 0, 1)$ [18]. *Timed automata* are dealt with, e.g., in [29,32,6,35].

Also the kind of specification considered can have different logical domains. We deal with *safety* properties: The violation of a safety property is expressed by stating that the last state of the computation does not fulfill the specification. Additional loop-determining techniques extend the method to *verify* properties for some problem classes (see e.g. [11,17]).

Once the existence of a counterexample of a fixed length is expressed by some formula, we need to check that formula for satisfiability: The formula is satisfiable if and only if the specification can be violated by a computation of that length. In the discrete case the check is carried out by a SAT-solver, i.e., a Boolean satisfiability checker, whereas in the mixed discrete-continuous case of hybrid and timed automata the check is usually done by combining a SAT- and an LP-solver (Linear Programming, see Section 5.2). Some popular solver are, e.g., ZChaff [28], BerkMin [23], MiniSAT [20], HySat [21],MathSAT [5], CVC Lite [8], and ICS [16]. Our approach, as introduced in the following sections, is not restricted to any fixed application domain. We illustrate its advantage by checking safety properties of some discrete systems (circuits) and of some linear hybrid automata.

One of our research goals within the AVACS project [7] is to improve the applicability of BMC to large hybrid automata. In an earlier paper [3] we concentrated on how BMC of linear hybrid automata can be accelerated by appropriate encoding of counterexamples as formulas, and by selective conflict learning. Those techniques were introduced in order to improve the *CPU running times*. We observed, however, that for some examples the real times needed were much longer than the CPU times. For long counterexamples the corresponding formulas are getting very large, as stated e.g. in [22]. Additionally, *learning* in the style of Shtrichman [30] considerably increases the memory consumption. When the memory requirements reach the computer's memory size, the computer starts to *swap*, thereby slowing down the computations by several orders of magnitude.

In this paper we discuss how the memory size necessary for solving a BMC problem can be reduced without increasing the running times of the solver. The main idea is to take advantage of the *symmetry* of BMC problems, and to store symmetric parts of the formulas in a parametric form. We introduce parametric data types for the internal solver structure and show that the usage of those parametric structures remarkably reduces the memory requirements of the solver. Experimental results show that the CPU times are not increased, and furthermore, due to lower demands on memory, swapping occurs much later resulting in shorter system times.

The paper is organized as follows: In Section 2 we review the BMC approach before introducing parametric datatypes in Section 3. Experimental results for circuits are presented in Section 4. Section 5 extends the results to linear hybrid automata. Finally, in Section 6 we discuss related work and draw conclusions.

2 Bounded Model Checking

Before presenting our work, we fi rst give a short review of discrete transition systems and of the encoding of their fi nite runs as fi rst-order predicate logic formulas, as introduced in BMC [10]. Furthermore, we describe relevant details of state-of-the-art solver for checking satisfi ability of such formulas.

2.1 Encoding Discrete Transition Systems

Below we formalize discrete transition systems. This kind of defi nition allows to deal with transition systems specifi ed by standard sequential circuits. On the other hand it can be extended to model linear hybrid systems.

Definition 2.1 [Discrete Transition System] A *discrete transition system (DTS)* is a tuple (V, L, I, T) with V a fi nite set of Boolean variables and L a fi nite set of nodes. We use $\mathcal{V}$ to denote the set of valuations $\nu : V \rightarrow \{0, 1\}$ and $\Sigma = (L \times \mathcal{V})$ to denote the set of states. The set $I \subseteq \Sigma$ defi nes the initial states, and $T \subseteq (L \times 2^{\mathcal{V}\times\mathcal{V}} \times L)$ specifi es the transition relation as a fi nite set of transitions with typical element t. We write $((l, \nu), (l', \nu')) \in t$ iff $t = (l, \mu, l')$ with $(\nu, \nu') \in \mu$. A ***run*** is a fi nite sequence $\sigma_0, \sigma_1, \ldots, \sigma_n$ of states such that $\sigma_0 \in I$ and $(\sigma_i, \sigma_{i+1}) \in t_i$ for some $t_i \in T$ for all $i = 0, \ldots, n-1$. A state is *reachable* if there is a run leading to it.

Since we deal with fi nite systems, the initial condition and the transitions of a DTS can be described by fi rst-order logic formulas $\mathit{Init}(s)$ and $\mathit{Trans}_t(s, s')$ for all $t \in T$, where s and s' explicitly denote the free variables occurring in the given formulas: $s = (v_0, \ldots, v_m)$ is the sequence of all variables and $s' = (v'_0, \ldots, v'_m)$ copies of them in order to describe the target valuation after a transition. Let furthermore $\mathit{Safe}(s)$ be a fi rst-order logic formula describing a safety property of the system. Counterexamples of a fi xed length k, i.e., runs of length k violating the property Safe, can be described by the following formula:

$$\varphi_k(s_0, \ldots, s_k) = \mathit{Init}(s_0) \wedge \Big(\bigwedge_{i=0,\ldots,k-1} \bigvee_{t\in T} \mathit{Trans}_t(s_i, s_{i+1})\Big) \wedge \neg\mathit{Safe}(s_k) \, .$$

Starting with $k = 0$ and iteratively increasing $k \in \mathbb{N}$, BMC checks whether φ_k is satisfi able. The algorithm terminates if φ_k is satisfi able, i.e., an unsafe state is reachable from an initial state in k steps.

2.2 Satisfiability Checking

The formulas φ_k describing counterexamples of length k are checked by a state-of-the-art DPLL (Davis-Putnam-Logemann-Loveland [15,14]) SAT-solver.

Before the satisfi ability check can start, the Boolean formula is transformed into a *conjunctive normal form* (CNF). In order to keep the formula as small as possible, auxiliary Boolean variables are used to build the CNF [34]. A formula in CNF-form is a conjunction of *clauses*, while each clause is the disjunction of *literals*.

We distinguish between positive and negative literals, being Boolean variables or their negations.

In order to satisfy the formula, each of the clauses must be satisfi ed, i.e., at least one of their literals must be true. The SAT-solver *assigns values* to the variables in an iterative manner. After each *decision*, i.e., free choice of an assignment, the solver *propagates* the assignment by searching for *unit-clauses* in that all literals but one are already false and thus the last unassigned literal is implied to be true.

If two unit-clauses imply different values for the same variable, a *conflict* occurs. In this case a conflict analysis takes place which results in *non-chronological backtracking* and *conflict learning* [9,27]. Intuitively, the solver applies resolution to some unit-clauses, using the implication tree, and inserts a new clause strengthening the problem constraints and restricting the state space for further search.

An important point for this paper is the usage of *watch-literals* for the detection of unit-clauses [28]. The basic idea is the following: If in a clause there are two unassigned (or already true) variables, then this clause cannot be a unit-clause. So it is enough to watch only two unassigned or true variables in each clause, which we call the watch-literals. If one of the watch-literals becomes false, we search for another literal in the clause, being unassigned or already true, and being different from the other watch-literal. Only if we cannot fi nd any new watch-literal, the clause is indeed a unit-clause. With this method, the number of clauses that we have to look at to determine the unit-clauses after a decision can be reduced remarkably.

3 Symmetries and Parametric Data Structures

In this main section we present how we make use of the inherent symmetries of BMC problems by parameterizing the solver-internal data structures.

3.1 Symmetries of BMC Problems

The formulas of BMC problems have a special structure: They describe computations, starting from an initial state, executing k transition steps, and leading to a state violating the specifi cation. Accordingly, the set of clauses generated by the SAT-solver can be grouped into clauses describing (1) the initial condition (*I-clauses*), (2) one of the transitions (*T-clauses*), and (3) the violation of the specifi - cation (*S-clauses*). The T-clauses can be further grouped into k sets describing the k computation steps. Those k T-clause sets describe the same transition relation, but at different time points. That means, they are actually the same up to renaming the variables. E.g., the 3rd iteration of a BMC problem could be represented by a clause set like this:

I-clauses	*T-clauses*	*S-clauses*
$(x_0 \vee y_0), \ldots$	$(x_0 \vee y_1 \vee \overline{z}_0), \ldots, (x_1 \vee \overline{y}_1 \vee z_0)$	$(y_3 \vee z_3), \ldots$
	$(x_1 \vee y_2 \vee \overline{z}_1), \ldots, (x_2 \vee \overline{y}_2 \vee z_1)$	
	$(x_2 \vee y_3 \vee \overline{z}_2), \ldots, (x_3 \vee \overline{y}_3 \vee z_2)$	

The T-clauses representing the 2nd transition step are the same as the T-clauses of the 1st step but v_i replaced by v_{i+1} for all variables v and indices i; we write $[v_{i+1}/v_i]$ for that substitution.

3.2 Parametric Data Structures

Since the T-clauses of different steps are the same up to variable renaming, it is enough to store a *parametric* version of a transition step, actually the transition relation, and remember the renaming in order to compute the information about the k different computation steps. If we need a clause of a certain transition step, for example to determine unit-clauses or for resolution, we just rename the variables in the parametric T-clauses accordingly. For the above example, we can store the parametric T-clause set $(x_0 \vee y_1 \vee \overline{z}_0), \ldots, (x_1 \vee \overline{y}_1 \vee z_0)$. The fi rst computation step is described by that clause set. Applying the substitution $[v_{i+1}/v_i]$ ($[v_{i+2}/v_i]$) gives the clause set describing the second (third) computation step.

In order to keep the solver structure simple, it is very important to use a fast and uncomplicated renaming mechanism. Look-up tables would be a possible solution, however, we expect that they would lead to increased computation times. Instead, we apply a more natural and easy naming convention, consisting of three stages:

- *Variables* are represented inside the solver not by an integer, but by a pair (a, i) of integers: the *abstract id* a identifi es a variable, and the *instance id* i the instance of the variable, i.e., the time instance at that the variable's value is considered. E.g., if x has the abstract id 5, then x in the initial state, i.e., x_0, is represented by $(5, 0)$, x after the fi rst transition step, i.e., x_1, by $(5, 1)$ and after the kth step for x_k we have $(5, k)$. Negation of a variable is expressed by the abstract id being negative. E.g., $\overline{x}_3$ is stored as $(-5, 3)$. Constants, being independent from the state they are evaluated in, have the instance id -1. In the following we treat constants as variables; if we say that we increase the instance id of a variable, then we mean that its instance id gets increased if it is non-negative, only.
- The contents of a clause, i.e., its *literals*, are now represented by a *list of integer pairs*. For example, the literals $(x_0, \overline{x}_1)$ are stored as $((5, 0), (-5, 1))$.
- Finally, each *clause* is referred to by a pair (a, i) of non-negative integers, where the *abstract id* a identifi es the parametric clause, usually by its index in the clause list, and the *instance id* i its instance. The ith instance of a parametric clause contains the literals of that clause with each (non-negative) instance id increased

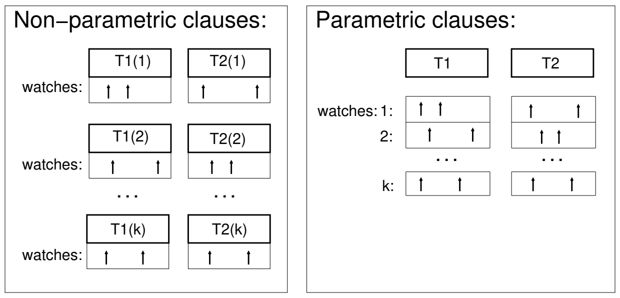


Fig. 1. Non-parametric and parametric data structures

by i. E.g., if the 7th parametric clause has literals $((5,0),(-5,1))$, then $(7,0)$ refers to the clause with literals $((5,0),(-5,1))$, whereas $(7,1)$ stands for the clause with the literals $((5,1),(-5,2))$, and $(7,k)$ for $((5,k),(-5,k+1))$.

In this way, dealing with parametric clauses becomes very simple: We store the literals of the T-clauses describing the fi rst computation step as parametric clauses. To compute the concrete literals of the T-clauses describing the ith computation step, we just increase the instance ids of all T-clause literals by $i-1$.

Above we described the encoding of the Boolean variables occurring in the formula. The representation of the auxiliary Boolean variables used to build the CNF effi ciently needs some more explanation: An auxiliary Boolean variable gets as instance id the smallest instance id occurring in the formula it encodes. The abstraction of the same formula at different time points use the same abstract id.

Note that parametric storage is possible only for the literals of the clauses. We still have to store the assignments for each variable instance on its own. Also the watch-literals of different instances of a parametric clause have to be stored separately. Thus, each parametric clause consists of a list of its (parametric) literals, and additionally a list of watch-pairs, determining the current watch-literals for each possible instance of the clause, as illustrated in Figure 1. The number of instances of a parametric clause is implicitly given by the length of the watch-pair list, and thus does not need to be stored explicitly. E.g., the parametric clauses of Figure 1 have k instances $1,\dots,k$, since they have k watch-pairs attached.

For conflict analysis, the solver stores the information, which unit-clause implied which assignment, in form of an implication tree. In the parametric approach, the implicating unit-clauses are identifi ed by an integer pair, as explained above.

Now, let us see how BMC works with the parametric structures. Initially, we check whether there are computations of length 0 or 1. At that point, the solver contains all I-clauses stating that the fi rst state is initial, all T-clauses describing the fi rst computation step, and all S-clauses stating that the last state in the run violates the specifi cation. For each subsequent BMC iteration we have to increment the computation length as follows:

- we add a new instance to each parametric T-clause by extending the watch-literal list by a new pair, and

- we increase the literals' instance ids in the S-clauses by 1.

The I-clauses remain untouched. Note that we do not need to insert any new clauses or literals for increasing the computation length! This is done simply by adding a new instance to the already existing transition clauses in the form of a new watch-pair. The number of clauses and the number of literals remain unchanged.

3.3 Conflict Learning

Besides clauses describing counterexamples we also have to pay attention to a second clause type: the conflict clauses. The conflict clauses learned during a SAT-check assure that the search does not enter the same search path (or similar search paths) again.

Usually, the conflict clauses learned during the SAT-check of a BMC instance get removed before checking the next BMC instance. However, they can also be partially re-used in the style of Shtrichman [31], thereby excluding search paths from the SAT-search already before the search starts: If a conflict clause is the result of a resolution applied to clauses that are present also in the next iteration, then the same resolution could be made in the new setting, too, and thus we can keep those conflict clauses. Furthermore, if all clauses used for resolution to generate a conflict clause are present in the next SAT iteration with an increased instance, then the same resolution could be made using the increased instances. Thus each such conflict clause can be added with an increased instance in the next BMC iteration. Accordingly, we distinguish between the following conflict clause types:

- *I-conflict clauses* result from resolution of I- and possibly T-(conflict-)clauses; they can be re-used in the next iterations, as those clauses are also present in all the following iterations, i.e., the same resolution could be made.
- *S-conflict clauses* result from resolution of S- and possibly T-(conflict-)clauses; they can be re-used with an increased instance only, as the instance of S-clauses gets increased in the next iteration.
- *T-conflict clauses* result from resolution of T-(conflict-)clauses, only; they can be re-used like I-conflict clauses and additionally inserted with an increased instance like S-conflict clauses, as all T-clauses are present in the next iteration both with the same and with an increased instance.

Note that conflict clauses stemming from both I- and S-clauses (*IS-conflict clauses*) cannot be re-used. Note furthermore that it is possible to learn even more than 2 instances of T-conflict clauses, if we record during the resolution not only which *kind* of clauses are involved (I, T, or S) but also which *instances* of T-clauses. However, our experiments show that learning all possible conflict clause instances leads to a large number of new clauses (or clause instances in the parametric case), each of which must be considered in the propagation of new decisions. That is the reason why learning too much rather slows down the SAT-check instead of accelerating it. We follow the policy of re-using conflict clauses when possible, and inserting T-conflict clauses additionally with one increased instance. This policy turned out to be successful within our experimental BMC framework.

We store conflict clauses in a parametric manner, too, analogously to the I-, T-, and S-clauses. After each iteration, additionally to the updates of the I-, T-, and S-clauses, the following updates take place:

- insert a new watch-pair for each T-conflict clause,
- increase the instance ids (if non-negative) of all literals in each S-conflict clause by 1, and
- delete all IS-conflict clauses.

Again, I-conflict clauses are untouched.

3.4 Variable Ordering

Our solver prototype uses a static variable order for selecting decision variables. As suggested in [31], the order is determined by the instance ids of the variables, and thus follows the natural temporal order of computation.

Nevertheless, our parametric data structures enable more variable-focused scoring heuristics like VSIDS [28], which do not handle the variables independently as pure CNF-SAT solver do, but group information belonging to several instances of one variable over the unfolded time-frames, allowing problem-oriented dynamic assignments.

4 Experimental Results

We implemented a SAT-solver, working mainly as described in Section 2.2, but with parametric internal data structures. To see the difference to the case without parametric structures, we created also a modifi ed solver, working exactly the same way but without parametric clauses. When a new BMC problem instance gets created, for the T-clauses and the T-conflict clauses the parametric solver adds a new clause instance by appending a new watch pair to the clause's watch list, while the solver without the parametric structure creates a new clause.

For the experiments we used a computer with an Intel Pentium $2,8$ GHz CPU and 1 GB of memory. Note, that the required memory is independent of the speed and memory size of the computer. However, if the memory size is below the requirements, swapping takes place which slows down the computation.

We applied BMC to check invariants of three benchmarks taken from the VIS benchmark suite [33] covering different application domains: `Am2910` (microcontroller), `Tcp` (communication protocol), and `UsbPhy` (Universal Serial Bus). Figure 2 shows the memory requirements: the heap peak during the iterations both for the non-parametric and for the parametric data structure is depicted.

Generally, using parametric clauses in the kth BMC iteration, the number of T-clauses can be reduced by the factor of k. T-conflict clauses learned in the iteration i get shifted in each iteration from $i+1$ to k by learning; instead of $k-i+1$ clauses we have to store only 1 parametric instance. The number of I- and S-clauses remains

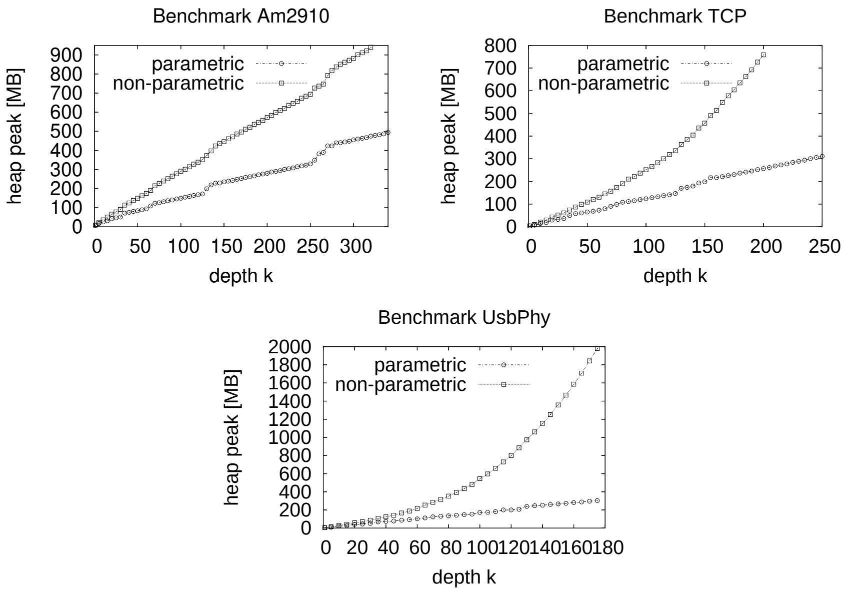


Fig. 2. Results for discrete VIS benchmarks

unchanged in both approaches; the same holds for I- and S-conflict clauses. It is worth to mention that the learned conflicts form a large part of the clauses.

The memory requirements cannot be reduced with the same factor as the number of clauses, since, e.g., the watch-literals must be stored for all clause instances. However, the memory requirements are still remarkably reduced. The degree of the reduction depends also on the size of the clauses.

The CPU times needed for the satisfi ability checks are approximately the same for the non-parametric and for the parametric solver (see Figure 6 for some experimental data). This is due to the natural data structures used to represent variables, literals, and clauses. Computing a certain concrete instance of a parametric clause is done by a few arithmetic additions.

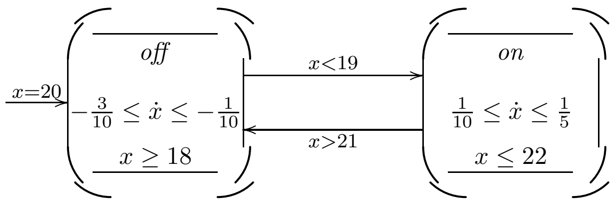
agr

5 Extension to Linear Hybrid Automata

The previously presented approach can be naturally extended to BMC of linear hybrid automata which is our primary goal as already mentioned in the introduction.

5.1 Linear Hybrid Automata

Hybrid automata [4,24] are a formal model to describe systems with combined discrete and continuous behavior. They are often illustrated graphically, like the one shown in Figure 3. This automaton models a thermostat, which senses the temperature x of a room and turns a heater on and off. When control stays in a location and time elapses, flow conditions in form of differential equations determine the



off
$x=20$
$-\frac{3}{10} \leq \dot{x} \leq -\frac{1}{10}$
$x \geq 18$
$x<19$
$x>21$
on
$\frac{1}{10} \leq \dot{x} \leq \frac{1}{5}$
$x \leq 22$

Fig. 3. Thermostat

continuous change of the real-valued variables. For example, in location *off* the temperature decreases according to the flow condition $-\frac{3}{10} \leq \dot{x} \leq -\frac{1}{10}$. Control may enter a location or stay in a location only as long as the location's invariant is satisfi ed. The invariant $x \geq 18$ of location *off* ensures that the heater turns on at latest when the temperature reaches 18 degrees. Control may move along a discrete jump from one location to another if the transition's condition is satisfi ed; additionally, the jump may cause discrete changes to the system state which is called the jump's effect. E.g., the transition from location *off* to *on* is enabled when the temperature is below 19 degrees; the temperature x does not change during the jump. Finally, an initial condition describes the starting point of the system's computations. For our example, initially the heater is *off* and the temperature is 20 degrees.

We consider the class of *linear hybrid automata* [4,24]. Applying BMC, counterexamples of a linear hybrid automaton can be encoded similarly to that of a DTS. In the hybrid case the underlying logic is the fi rst-order logic over $(\mathbb{R}, +, <, 0, 1)$, i.e., formulas are the Boolean combinations of (in)equations over linear terms using real-valued variables. The transition relation captures two cases: discrete jumps and continuous flows, that must both be represented in the BMC encodings. For a detailed description of the encodings and optimizations see [3].

5.2 LP-SAT-Checking

The above formulas describing counterexamples of a fi xed length are checked, like in the discrete case, by a suitable solver. As now we are dealing with the Boolean combination of linear (in)equations over real-valued variables, the satisfi - ability check is done by a combined SAT-LP-solver, as illustrated in Figure 4.

First, the hybrid formulas are abstracted in an over-approximative manner to pure Boolean ones by replacing each real constraint, i.e., each linear (in)equation, by an auxiliary Boolean abstraction variable. This Boolean abstraction is checked for satisfi ability by a SAT-solver. In case the abstraction is unsatisfi able, the concrete hybrid formula is unsatisfi able, too. Otherwise, if the abstraction has a solution, then the LP-solver checks whether there is a corresponding solution in the real domain. I.e., the LP-solver collects all those real constraints whose abstraction variables are true and the negation of all those whose abstraction variables are false, and checks whether they are together satisfi able using a Simplex-based approach similar to [21]. If yes, then we have found a solution for the concrete problem. If not, then the LP-solver provides an explanation in the form of an unsatisfi able

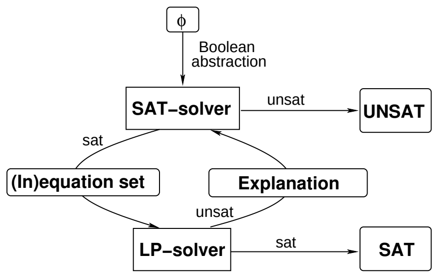


Fig. 4. Basic structure of combined SAT-LP-solver

(in)equation set that explains the contradictory assignment within the real domain. The SAT-solver can now refine the abstraction by excluding the abstracted explanation in the further search.

The above mechanism is known as *lazy* satisfiability check. *Less lazy* variants check for consistency in the real domain more often, not only for full Boolean solutions, but also for partial ones. This allows earlier detection of real conflicts, and thus also earlier backtracking for such conflicts. Though LP-checks are relatively expensive in running time, the advantage of earlier backtracking usually pays off. However, the degree of laziness is crucial for the running time. If there are only few solutions for the abstraction, then the full lazy variant will probably be faster, while for abstractions with many solutions the less lazy variant is expected to be more efficient. In our solver, the frequency of LP-checks is determined dynamically depending on the number of solutions already found for the abstraction.

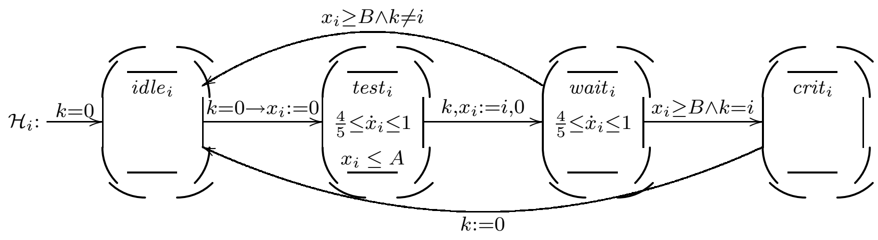
During the SAT-checks, our solver also learns the explanations served by the LP-solver in order to refine the abstraction. Those explanations are contradictions in the real-valued domain, thus we could exclude them using all possible renamings of the involved real-valued variables. In our solver those conflict clauses, stemming from the real-valued domain, are treated as T-conflict clauses.

5.3 Results

We also implemented a combined SAT-LP-solver, working as the SAT-solver of the previous section, but extended with an LP-solver for the real part of the check. Similarly to the discrete case, we compare a parametric and a non-parametric version of the solver, using the same SAT-LP-algorithm.

The experiments were carried out on the same computer as in the discrete case. We used as first example Fischer's mutual exclusion protocol [26] for 3 and for 4 processes (see Figure 5 for the ith process). The specification states the mutual exclusion property, i.e., that at each time point there is at most one process in its critical section. The second example is a Railroad Crossing [24], consisting of 3 parallel automata: one modeling a train, one a railroad crossing gate, and one a controller. The specification requires that the gate is always fully closed when the train is near to the railroad crossing.

Figure 6 shows the running times for Fischer's protocol with 3 processes (on the



$\mathcal{H}_i$:
$k{=}0$
$idle_i$
$k{=}0{\rightarrow}x_i{:=}0$
$test_i$
$\frac{4}{5}{\leq}\dot{x}_i{\leq}1$
$x_i \leq A$
$k{,}x_i{:=}i{,}0$
$wait_i$
$\frac{4}{5}{\leq}\dot{x}_i{\leq}1$
$x_i{\geq}B{\wedge}k{=}i$
$crit_i$
$x_i{\geq}B{\wedge}k{\neq}i$
$k{:=}0$

Fig. 5. Fischer's mutual exclusion protocol: The ith process

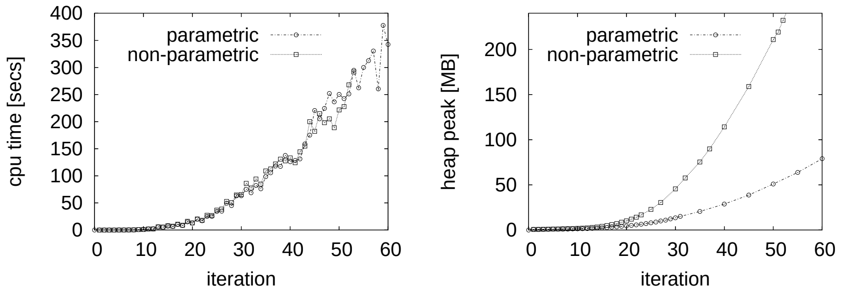


Fig. 6. Results for Fischer protocol with 3 processes

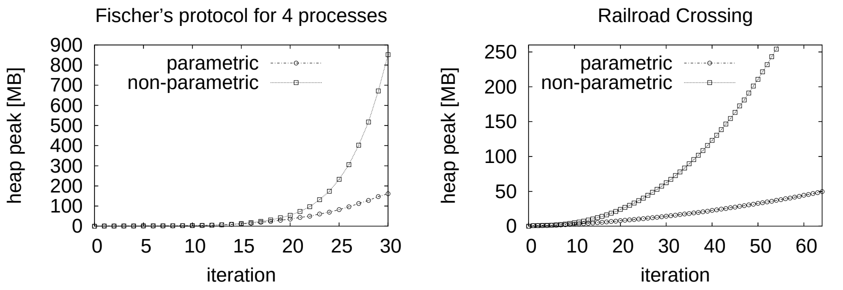


Fig. 7. Results for the Fischer protocol with 4 processes and the Railroad example

left), and the memory requirements (on the right) compared to the non-parametric version of our solver. The running times show that the computation is not slowed down by the parametric structures. Figure 7 shows the memory consumption for the remaining examples, again for both, the non-parametric and parametric version.

6 Conclusion and Related Work

In this paper we introduced parametric data structures to reduce the memory requirements of satisfiability checking for the special purpose of bounded model checking. The application of BMC to some discrete and hybrid examples served to point out the practical relevance of our approach.

Most research on SAT-solving is done in the important area of increasing the runtime effi ciency. Related work, like those dealing with the basic solver algorithms, bounded model checking, and learning in the context of BMC etc., is already mentioned in the introduction.

We know of only two papers explicitly dealing with the reduction of the BMC memory requirements. In [19], similarly to our approach, the authors make use of the symmetry of the transition steps. However, instead of introducing new internal data structures as we do, they apply quantifi cation to compress the k transitions of a counterexample description into a single quantifi ed term. The quantifi ed formula is checked for satisfi ability by a dedicated QBF solver.

The approach of [22] tackles memory problems during BMC by distributed computation. There, the unfolding of the clause set is partitioned and each partition is assigned to one component in the network. The focus lies on the distribution of the Boolean constraint propagation to local components such that a memory reduction is achieved due to the decentralized organization. Thus [22] works in some sense orthogonal to our approach where we exploit the inherent symmetry of the BMC formula by means of parametric data structures. As to future work, we are also working on a parallelization scheme that incorporates both ideas.

Another interesting point is the integration of optimization techniques like cone-of-influence reduction [12] and don't-care optimization [25]. While the former does not limit our concept of parameterization, the latter requires a feasibility study as future work.

Acknowledgements

We thank Ralf Wimmer and Jochen Eisinger for their valuable comments on the paper and Christian Herde and Martin Fränzle for the fruitful discussions.

References

[1] "CADE'02," LNAI **2392**, Springer-Verlag.

[2] "CAV'04," LNCS **3114**, Springer-Verlag.

[3] Ábrah´am, E., B. Becker, F. Klaedke and M. Steffen, *Optimizing bounded model checking for linear hybrid systems*, in: *Proc. of VMCAI'05*, LNCS **3385**, pp. 396–412.

[4] Alur, R., C. Courcoubetis, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis and S. Yovine, *The algorithmic analysis of hybrid systems*, Theoretical Computer Science **138** (1995), pp. 3–34.

[5] Audemard, G., P. Bertoli, A. Cimatti, A. Korniłowicz and R. Sebastiani, *A SAT based approach for solving formulas over boolean and linear mathematical propositions*, in: *Proc. of CADE'02* [1].

[6] Audemard, G., A. Cimatti, A. Korniłowicz and R. Sebastiani, *Bounded model checking for timed systems*, in: *Proc. of FORTE'02*, LNCS **2529**, pp. 243–259.

[7] *Transregional collaborative research center 14 AVACS: Automatic Verification and Analysis of Complex Systems*, `http://www.avacs.org`.

[8] Barrett, C. and S. Berezin, *CVC Lite: A new implementation of the cooperating validity checker*, in: *Proc. of CAV'04* [2], pp. 515–518.

[9] Bayardo Jr., R. and P. Schrag, *Using CSP look-back techniques to solve real-world SAT instances*, in: *National Conference on Artificial Intelligence (AAAI)*, 1997.

[10] Biere, A., A. Cimatti, E. Clarke and Y. Zhu, *Symbolic model checking without BDDs*, in: *Proc. of TACAS'99*, LNCS **1579**, pp. 193–207.

[11] Biere, A., A. Cimatti, E. M. Clarke, O. Strichman and Y. Zhu, *Bounded model checking*, Advances in Computers **58** (2003).

[12] Biere, A., E. Clarke, R. Raimi and Y. Zhu, *Verifying safety properties of a PowerPC*$^{\mathrm{TM}}$ *microprocessor using symbolic model checking without BDDs*, in: *Proc. of CAV'99*, LNCS **1633**.

[13] Copty, F., L. Fix, R. Fraer, E. Guinchiglia, G. Kamhi and M. Y. Vardi, *Benefits of bounded model checking in an industrial setting*, in: *Proc. of CAV'01*, LNCS **2102**, pp. 436–453.

[14] Davis, M., G. Logemann and D. Loveland, *A machine program for theorem-proving*, Communications of the ACM **5** (1962), pp. 394–397.

[15] Davis, M. and H. Putnam, *A computing procedure for quantification theory*, Journal of the ACM **7** (1960), pp. 201–215.

[16] de Moura, L. and H. Rueß, *An experimental evaluation of ground decision procedures*, in: *Proc. of CAV'04* [2], pp. 162–174.

[17] de Moura, L., H. Rueß and M. Sorea, *Bounded model checking and induction: From refutation to verification*, in: *Proc. of CAV'03*, LNCS **2725**, pp. 14–26.

[18] de Moura, L., H. Rueß and M. Sorea, *Lazy theorem proving for bounded model checking over infinite domains*, in: *Proc. of CADE'02* [1], pp. 438–455.

[19] Dershowitz, N., Z. Hanna and J. Katz, *Bounded model checking with QBF*, in: *Proc. of SAT'05*, LNCS **3569**, pp. 408–414.

[20] E´en, N. and N. Sörensson, *An extensible SAT-solver.*, in: *Proc. of SAT'03*, LNCS **2919**, pp. 502–518.

[21] Fränzle, M. and C. Herde, *Efficient proof engines for bounded model checking of hybrid systems.*, ENTCS **133** (2005), pp. 119–137.

[22] Ganai, M. K., A. Gupta, Z. Yang and P. Ashar, *Efficient distributed SAT and SAT-based distributed bounded model checking*, in: *Proc. of CHARME'03*, LNCS **2860**, pp. 334–347.

[23] Goldberg, E. and Y. Novikov, *BerkMin: A fast and robust SAT-solver*, in: *Proc. of DATE'02*, pp. 142–149.

[24] Henzinger, T. A., *The theory of hybrid automata*, in: *Proc. of LICS'96*, pp. 278–292.

[25] Kuehlmann, A., *Dynamic transition relation simplification for bounded property checking.*, in: *Proc. of ICCAD'04*, pp. 50–57.

[26] Lynch, N., "Distributed Algorithms," Kaufmann Publishers, 1996.

[27] Marques-Silva, J. and K. Sakallah, *GRASP: A search algorithm for propositional satisfiability*, IEEE Trans. on Comp. **48** (1999), pp. 506–521.

[28] Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Yang and S. Malik, *Chaff: Engineering an efficient SAT solver*, in: *Proc. of DAC'01*, pp. 530–535.

[29] Niebert, P., M. Mahfoudh, E. Asarin, M. Bozga, N. Jain and O. Maler, *Verification of timed automata via satisfiability checking*, in: *Proc. of FTRTFT'02*, LNCS **2469**.

[30] Shtrichman, O., *Pruning techniques for the SAT-based bounded model checking problem*, in: *Proc. of CHARME'01*, LNCS **2144**, pp. 58–70.

[31] Shtrichman, O., *Accelerating bounded model checking of safety formulas*, Formal Methods in System Design **24** (2004), pp. 5–24.

[32] Sorea, M., *Bounded model checking for timed automata*, ENTCS **68** (2002).

[33] The VIS Group, *VIS: A system for verification and synthesis*, in: *Proc. of CAV'96*, LNCS **1102**, pp. 428–432, see also `http://vlsi.colorado.edu/∼vis`.

[34] Tseitin, G., *On the complexity of derivations in propositional calculus*, in: *Studies in Constructive Mathematics and Mathematical Logics*, 1968 .

[35] Wo´zna, B., A. Zbrzezny and W. Penczek, *Checking reachability properties for timed automata via SAT*, Fundamenta Informaticae **55** (2003), pp. 223–241.

Interpolant Learning and Reuse in SAT-Based Model Checking

Joao Marques-Silva [1]

School of Electronics and Computer Science
University of Southampton, Southampton, UK

Abstract

One of the most paradigmatic practical applications of Boolean Satisfiability (SAT) is bounded model checking (BMC). The utilization of SAT in model checking has allowed significant performance gains and, as a consequence, a large number of commercial verification tools now include SAT-based model checkers. Recent work has provided SAT-based BMC with completeness conditions, and this is generally referred to as unbounded model checking (UMC). Among the existing approaches for SAT-based UMC, the utilization of interpolants is among the most effective. Despite their success, interpolants have only been used for identifying a fixed point of the set of reachable states. This paper extends the utilization of interpolants in SAT-based model checking. This is achieved by observing that, under reasonable assumptions, interpolants can be *reused*, i.e. computed interpolants can be reused at later stages of the model checking process. The paper develops conditions for validity of interpolant reuse. Preliminary practical experience on interpolant learning and reuse is reported.

Key words: Boolean Satisfiability, Bounded Model Checking, Interpolants.

1 Introduction

The utilization of Boolean Satisfiability (SAT) in Model Checking has been the subject of extensive research in recent years. The main result of this effort has been a number of fairly competitive incomplete and complete SAT-based model checking algorithms [3,4,5,20,21,25,26]. Moreover, SAT-based model checking has also been rapidly adopted by industry, and a number of vendors have included SAT-based Model Checking in their tools.

The utilization of SAT in model checking was first proposed in the form of Bounded Model Checking (BMC) [3], where a counterexample is searched for

[1] Email:jpms@ecs.soton.ac.uk

increasing unfoldings of a finite state automaton. The original BMC work has been shown to be extremely useful for finding counter-examples but, unless the recurrence (or the reachability) diameter of the automaton is known [2], the BMC procedure is incomplete.

Different solutions have been proposed for ensuring the completeness of BMC [25,5,17,16,21], the most promising of which is arguably based on the utilization of interpolants [21].

This paper reviews the utilization of interpolants in SAT-based unbounded model checking and proposes the learning and reuse of computed interpolants with the purpose of allowing increased search pruning for subsequent calls to the SAT solver during the model checking process. The paper shows that different interpolants can be computed and used in different contexts.

The paper is organized as follows. The next section provides a necessarily brief perspective on SAT solvers and related concepts. Afterwards, Section 3 reviews SAT-based model checking, including bounded and unbounded model checking. Section 4 develops conditions for reusing learnt interpolants. Initial practical experience is summarized in Section 5 and Section 6 concludes the paper.

2 Preliminaries

Propositional formulas are defined over finite sets of Boolean variables $X = \{x_1, x_2, \ldots\}$, $W = \{w_1, w_2, \ldots\}$, X_1, X_2, etc., where each variable can be assigned value 1 (True) or 0 (False). In what follows propositional formulas are represented by $\psi_1, \psi_2, \ldots$. When relevant other subscripts can be used, e.g. ψ_a, ψ_b, etc. For specific cases, letters and names representing predicates are also used for denoting the associated propositional formulas, examples include I, T, F, P, Q and Bmc. When referring to propositional formulas in conjunctive normal form (CNF), we associate with each propositional formula $\psi_a(X_a)$ a CNF formula $\varphi_a(X_a, U_a)$, where U_a denotes a set of auxiliary Boolean variables. Formulas in CNF consist of a conjunction of clauses (each clause represented by ω_i), where each clause consists of a disjunction of literals (represented by l_j). When used in an expression, a propositional formula ψ is interpreted as a predicate, and so corresponds to $\psi = 1$. Similarly, when the propositional formula $\neg\psi$ is used in an expression, it corresponds to $\psi = 0$.

We consider model checking of LTL safety properties G ψ_S. A finite state automaton $M = (I, T, F)$ is assumed, where I is a predicate defined on state variables, T is the state transition relation, and F represents the failing property (i.e. $F = \neg\psi_S$), defined on state variables. Moreover, the utilization of predicates I, T or F assumes an underlying automaton $M = (I, T, F)$. As mentioned above, for simplicity, the propositional formulas associated with these predicates are represented with the same letters, I, T and F.

It will also be necessary to map propositional formulas from one set of variables to another set of variables. The notation $\psi(Y/Y_k)$ is used to denote that the propositional formula ψ, defined over the set of variables Y, is mapped into the set of variables Y_k. Moreover, state variables are preferably represented as set Y, Y_k when referring to the state variables in time step k, Boolean circuit variables are preferably represented as sets X or W, respectively X_k and W_k for variables in time step k, and finally auxiliary variables used in the CNF representation are preferably represented as sets W or Z.

2.1 Boolean Satisfiability Solvers

The remarkable evolution of Boolean Satisfiability (SAT) solvers over the last decade [19,23,14] has motivated the application of SAT in model checking. The most effective SAT solvers are based on backtrack search [9] and share a number of key techniques, including:

- Unit clause rule, also referred as Boolean constraint propagation, that consists of the identification of implied variable assignments [10].
- Clause learning, consisting of learning new clauses in presence of conflicts during the execution of backtrack search. A few techniques related with clause learning are the utilization of unique implication points (UIPs) [19], and non-chronological backtracking [19].
- Memory efficient lazy data structures [23].
- Adaptive branching heuristic, usually derived from the VSIDS heuristic [23].
- Utilization of search restarts [15], by using some completeness criterion.

Because modern backtrack search SAT solvers learn clauses, it is straightforward to track all the learned clauses, and use these clauses for constructing a resolution refutation (or unsatisfiability proof) of the original formula [28].

2.2 SAT-Related Concepts

This subsection addresses a number of byproducts of modern SAT solvers, which are required for the utilization of interpolants in SAT-based model checking. For this purpose, we review *proof traces*, *unsatisfiable cores* and *unsatisfiability proofs*.

As mentioned above, modern SAT solvers learn clauses. For unsatisfiable instances, the original clauses and the learned clauses can be used for generating a resolution-based unsatisfiability proof [28]. Modern SAT solvers can be instructed for generating a *proof trace*, which associates with each learned clause ω, all the clauses that explain the creation of ω [28].

Given a proof trace Γ, where the final traced clause is the empty clause $\perp$, we can identify, in linear time on the size of the proof trace, a subset of the original set of clauses which is itself unsatisfiable [28]. This subset is referred to as an *unsatisfiable core*.

Moreover, and given a proof trace Γ, generated by a SAT solver, it is possible to create a resolution-based unsatisfiability proof in time and size linear on the size of the proof trace.

Definition 2.1 [Unsatisfiability Proof [21]] A proof of unsatisfiability Π for a set of clauses φ is a directed acyclic graph (V_Π, E_Π), where V_Π is a set of clauses, such that:

- For every $\omega \in V_\Pi$, either
  - $\omega \in \varphi$, and ω is a root, or
  - ω has two predecessors, ω_1 and ω_2, such that ω is the resolvent of ω_1 and ω_2 (the variable v used for resolving ω_1 with ω_2 is referred to as the *pivot* variable of the resolution step), and
- the empty clause $\perp$ is the unique leaf.

2.3 Craig Interpolants

Assume a propositional formula $\psi_A(Y, X)$, defined over the sets of variables Y and X, and a propositional formula $\psi_B(Y, W)$, defined over the sets of variables Y and W. If $\psi_A(Y, X) \land \psi_B(Y, W)$ is unsatisfiable, then there exists a propositional formula $\psi_P(Y)$, defined over the set of variables Y, such that $\psi_A(Y, X) \to \psi_P(Y)$ is a tautology and $\psi_B(Y, W) \land \psi_P(Y)$ is unsatisfiable. The propositional formula $\psi_P(Y)$ is referred to as an *interpolant* for $\psi_A(Y, X)$ and $\psi_B(Y, W)$ [8]. Recent work has shown that an interpolant can be constructed in linear time on the size of a resolution refutation of $\psi_A(Y, X) \land \psi_B(Y, W)$ [24].

In what follows we outline McMillan's interpolant construction [21], even though Pudlák's construction [24] could also be considered. Regarding the propositional formulas $\psi_A(Y, X)$ and $\psi_B(Y, W)$, and associated CNF formulas, respectively $\varphi_A(Y, X, U)$ and $\varphi_B(Y, W, V)$, variables in set Y are referred to as *global* variables, whereas variables in sets X and U are *local* to $\varphi_A(Y, X, U)$, and the variables in sets W and V are *local* to $\varphi_B(Y, W, V)$. Further, let $g(\omega)$ denote the literals corresponding to global variables in clause ω.

Definition 2.2 [Interpolant [21]] Let (φ_A, φ_B) be a pair of clause sets and let Π be a proof of unsatisfiability of $\varphi_A \cup \varphi_B$, with leaf vertex $\perp$. For each vertex $\omega \in V_\Pi$, let ψ_ω be a Boolean formula, such that:

- If ω is a root then
  - if $\omega \in \varphi_A$ then $\psi_\omega = g(\omega)$,
  - else $\psi_\omega = \textsc{True}$
- else, let ω_1, ω_2 be the predecessors of ω and let v be their pivot variable
  - if v is local to φ_A, then $\psi_\omega = \psi_{\omega_1} \lor \psi_{\omega_2}$,
  - else $\psi_\omega = \psi_{\omega_1} \land \psi_{\omega_2}$

The Π-interpolant of (φ_A, φ_B), denoted $\textsc{Itp}(\Pi, \varphi_A, \varphi_B)$ is $\psi_\perp$.

The interpolant $\textsc{Itp}(\Pi, \varphi_A, \varphi_B)$ has size linear on the size of the unsatisfiability proof [24,21].

Algorithm 1 Organization of BMC

```
BMC($M = (I, T, F)$, $\lambda$, $\iota$, $\mu$)
1  $j \leftarrow 0$
2  $k \leftarrow \lambda$
3  while $k \leq \mu$
4        do $\varphi \leftarrow$ CNF(BMC$_j^k(M), W$)
5           if SAT($\varphi$)
6              then return false ▷ Found counterexample
7           $k \leftarrow k + \iota$
8  return true
```

3 SAT-Based Model Checking

This section overviews the work on using SAT in model checking, emphasizing the initial work on Bounded Model Checking (BMC) and the more recent work on Unbounded Model Checking (UMC).

3.1 Bounded Model Checking

The generic Boolean formula associated with SAT-based BMC is the following [3,26,2]:

$$\textsc{Bmc}_j^k(M) = I(Y_0) \wedge \left(\bigwedge_{0 \leq i < k} T(Y_i, Y_{i+1}) \right) \wedge \left(\bigvee_{j \leq i \leq k} F(Y_i) \right) \tag{1}$$

This formula represents the unfolding of the state machine for k time steps, where $I(Y_0)$ represents the initial state, $T(Y_i, Y_{i+1})$ represents the transition relation between states Y_i and Y_{i+1}, and $F(Y_i)$ represents the failing property in time step i. Given the Boolean formula $\textsc{Bmc}_j^k(M)$, it is straightforward to generate a CNF formula φ, by applying Tseitin's transformation [27] and by using additional auxiliary Boolean variables. This formula can then be evaluated by a SAT solver.

The typical organization of BMC for safety properties is illustrated in Algorithm 1. The details regarding the sets of variables associated with each propositional formula are omitted, but are clear from the context. Experimental evidence has confirmed SAT-based BMC to be an extremely competitive technique, that has been widely applied in industrial settings [2,12,7].

In order to describe the work on UMC and the reusing of interpolants, the following predicates are extensively used:

$$\textsc{Unfold}_r^s(M) = I(Y_{-r}) \wedge \left(\bigwedge_{-r \leq i < s} T(Y_i, Y_{i+1}) \right) \tag{2}$$

$$\textsc{Tran}_s^t(M) = \bigwedge_{s \leq i < t} T(Y_i, Y_{i+1}) \tag{3}$$

$$\textsc{Prop}_v^u(M) = \left(\bigwedge_{u \le i < u+v} T(Y_i, Y_{i+1})\right) \wedge \left(\bigvee_{u \le i \le u+v} F(Y_i)\right) \tag{4}$$

Hence, we can express the BMC formula in terms of these predicates:

$$\begin{aligned} \text{BMC}_j^k(M) &= \textsc{Unfold}_0^j(M) \wedge \textsc{Prop}_{k-j}^j(M) \\ &= \textsc{Unfold}_0^0(M) \wedge \textsc{Tran}_0^j(M) \wedge \textsc{Prop}_{k-j}^j(M) \end{aligned} \tag{5}$$

3.2 Unbounded Model Checking

A key difficulty with BMC is its inability for proving that there is no counterexample for a given safety property G ψ_S. Unless the recurrence (or the reachability) diameter [2] of an automaton is known, it is not possible to establish the value of the upper bound (UB) used in Algorithm 1; in the case the recurrence diameter is known, BMC becomes complete. In general the recurrence diameter of an automaton is not known, and so BMC is incomplete. As a result, in recent years different approaches have been proposed for ensuring the completeness of SAT-based model checking. We refer to these approaches as *Unbounded Model Checking* (UMC) [20,21]. The first UMC SAT-based approach was proposed by Sheeran et al. in [25] and extended in [4]. Additional techniques include [5,20,13,22,21,16]. The induction-based approach of Sheeran et al. [25] requires unfolding the state machine for the largest simple path between any two reachable states in the worst case. However, the largest simple path between any two reachable states can be exponentially larger than the reachability diameter. Alternatively, Chauhan et al. [5] and Glusman et al. [13] propose refinement techniques based on elimination of false counterexamples. Another approach based on iterative abstraction is proposed by Gupta et al. in [16]. More recently, McMillan and Amla [22] propose the utilization of proof-based abstraction, even though the proposed approach is not fully SAT-based. According to experimental data from [21], the utilization of interpolants in SAT-based model checking is the most effective approach. We detail the utilization of interpolants in the next section.

3.3 Interpolant-Based Unbounded Model Checking

Recent work on SAT-based Unbounded Model Checking has addressed the utilization of interpolants [21], with quite promising experimental results. This section reviews McMillan's interpolant-based UMC algorithm [21].

The definition of the BMC proposition formula is modified slightly with respect to (1):

$$\begin{aligned} \textsc{Pref}_l(M) &= I(Y_{-l}) \wedge \left(\bigwedge\nolimits_{-l \le i < 0} T(Y_i, Y_{i+1})\right) \\ &= \textsc{Unfold}_l^0(M) \end{aligned} \tag{6}$$

Algorithm 2 UMC Algorithm

```
UMC($M = (I, T, F)$)
1   $k \leftarrow 0$
2   if SAT($I \wedge F$)
3      then return false ▷ Counterexample found
4           while true
5              do $status$ = CHECKFIXPOINT($M, k$)
6                 if status = false
7                    then return false ▷ Counterexample found
8                    else  if status = true
9                             then return true ▷ Property proved
10                k ← k + 1 ▷ Unfold further
```

$$\begin{aligned}\text{(7)}\qquad \textsc{Suff}_j^k(M) &= \left(\textstyle\bigwedge_{0\le i<k} T(Y_i, Y_{i+1})\right) \wedge \left(\textstyle\bigvee_{j\le i\le k} F(Y_i)\right) \\ &= \textsc{Tran}_0^j(M) \wedge \textsc{Prop}_{k-j}^j(M)\end{aligned}$$

Hence, the BMC formula becomes:

$$\text{(8)}\qquad \textsc{Bmc}_j^k(M) = \textsc{Pref}_1(M) \wedge \textsc{Suff}_j^k(M)$$

The above equation corresponds to the one proposed by McMillan [21], where the separation between prefix and suffix identifies the set of variables with respect to which interpolants are to be computed.

The SAT-based model checking algorithm can be organized into two main phases: a BMC loop, where the circuit is unfolded, and a fixed point checking step, that checks for the existence of a counterexample and where the existence of a fixed-point is tested. Observe that the second phase requires the iterative computation of interpolants until a fixed-point is reached or a true or (possibly) false counterexample is identified. The organization of the BMC loop is outlined in Algorithm 2, whereas the organization of fixed point checking step is outlined in Algorithm 3.

For the BMC loop there is no upper bound on the number of unfoldings, since the algorithm is now complete. The increment of k is not required to be 1. In fact, feeback from the fixed point checking procedure can be used for increasing k by values larger than 1 [18]. In addition, observe that the fixed point checking procedure consists of iterative computation of interpolants, where for iteration m the interpolant represents an abstraction of the reachable states in m time steps [21]. At each iteration of the UMC fixed point checking procedure, the existence of a fixed-point is tested. The fixed-point is reached when the abstraction of the reachable states in m time steps contains only states already included in the abstractions of the reachable states in less than m time steps. Finally, observe that the algorithm sets $j = 0$, because interpolants are computed with respect to Y_0.

Algorithm 3 Fixed point identification in SAT-based UMC

```
CheckFixpoint($M = (I, T.F), k$)
1   $R \leftarrow I$
2   while true
3       do $M' \leftarrow (R, T, F)$
4          $A \leftarrow$ Cnf(Pref$_1(M'), W_1$)
5          $B \leftarrow$ Cnf(Suff$_0^k(M'), W_2$)
6          $(isSat, \Gamma) \leftarrow$ Sat($A \cup B$)
7          if $isSAT$
8             then if $R = I$
9                     then return false
10                    else  return abort
11         $\triangleright$ $A \cup B$ is unsat
12         $\Pi \leftarrow$ UnsatProof($\Gamma$)
13         $P \leftarrow$ Itp($\Pi, A, B$)
14         $R' \leftarrow P(Y/Y_0)$
15         $C \leftarrow$ Cnf($\neg R, W_3$)
16         $D \leftarrow$ Cnf($R', W_4$)
17         $(isSat, \Gamma) \leftarrow$ Sat($C \cup D$)
18         if not $isSAT$
19            then return true
20         $R \leftarrow R \vee R'$
```

4 Interpolant Learning and Reuse

This section develops conditions for reusing computed interpolants, and consists of two main parts. Conditions for interpolants representing over-approximations of the set of reachable states, and conditions for interpolants representing over-approximations of the set of states satisfying the failing property. We should note that the work on interpolant reuse is largely motivated by previous (and successful) work on clause reuse [26]. Clause reuse has been used extensively in BMC and is widely regarded as a key technique [26,12].

The main motivation is to develop conditions which enable computed interpolants to be reused. Hence, the following definition is used extensively.

Definition 4.1 A Boolean formula ψ_N is said to be *usable* for Boolean formula ψ_B iff $\psi_B \rightarrow \psi_N$.

Hence, ψ_N preserves satisfiability of the original formula and so we get the following straightforward result:

Proposition 4.2 *Let ψ_N be usable for ψ_B. Then ψ_B is satisfiable iff $\psi_B \wedge \psi_N$ is satisfiable.*

In order to generalize the computation of interpolants, equation (5) is modified as follows:

$$\textsc{Bmc}_j^k(M) = \textsc{Unfold}_0^k(M) \wedge \textsc{Prop}_j^k(M) \tag{9}$$

Observe that the new equation differs from (5) and (8). In equation (9) the failing property is checked for only in the last j time steps for an unfolding of $k + j$ time steps [2]. (This approach is also used for example in [7,25,12]). For simplicity we assume $j = 0$; generalization for $j > 0$ is simple.

The standard interpolants used in [21] are referred to as *direct interpolants*. It is also possible to compute *reverse interpolants* by exchanging the sets A and B in the definition of interpolant. *Direct* interpolants are computed as described in McMillan's work [21] (see also the previous section), but relaxing the 1 time step unfolding for A. For computing an interpolant after r time steps from I and $t = k - r$ time steps from F, the propositional formulas for A and B become:

$$A = \textsc{Cnf}(\textsc{Unfold}_0^r(M), W_1) \tag{10}$$

$$B = \textsc{Cnf}(\textsc{Tran}_{k-t}^k(M) \wedge \textsc{Prop}_0^k(M), W_2) \tag{11}$$

The interpolant computed with A and B above will be denoted P_t^r. It is also possible to compute an interpolant by replacing I with another interpolant P_v^u:

$$A = \textsc{Cnf}(P_v^u(Y_0) \wedge \textsc{Tran}_0^r(M), W_1) \tag{12}$$

$$B = \textsc{Cnf}(\textsc{Tran}_{k-t}^k(M) \wedge \textsc{Prop}_0^k(M), W_2) \tag{13}$$

And the new interpolant is denoted P_t^{u+r}.

Reverse interpolants are computed by interchanging A and B in the definitions above, and will be denoted respectively by Q_t^r and Q_t^{u+r}.

Consequently, P_t^r, $r, t \geq 0$, denotes the direct interpolant computed with a (possibly virtual) unfolding of r time states from the initial state, and t time steps until the failing property is checked for. Hence, P_t^r represents an *over-approximation* of the set of states reachable in r time steps and an *under-approximation* of the set of states which do not satisfy the failing property in t time steps. Similarly, Q_t^r, $r, t \geq 0$, denotes the reverse interpolant computed with a (possibly virtual) unfolding of r time states from the initial state, and t time steps until the failing property is checked for. Hence, Q_t^r represents an *under-approximation* of the set of states that are not reachable in r time steps and an *over-approximation* of the set of states which satisfy the failing property in t time steps.

Given the definitions of direct and reverse interpolants, we can now establish conditions for interpolant reuse in SAT-based model checking.

Theorem 4.3 *Let* $\textsc{Bmc}_j^k(M)$ *be given by (9). Then the following holds:*

(i) $P_t^r(Y_r)$ *is usable for* $\textsc{Bmc}_j^k(M)$, *with* $t \geq 0$ *and* $0 \leq r \leq k$.

(ii) $\neg P_t^r(Y_{k-t})$ *is usable for* $\textsc{Bmc}_j^k(M)$, *with* $r \geq 0$ *and* $0 \leq t \leq k$.

[2] The automaton is assumed to be stuttering closed [6,21].

Proof.

(i) If $\textsc{Bmc}_j^k(M)$ is satisfiable, then $\textsc{Unfold}_0^r(M)$, with $r \leq k$ is also satisfiable and Y_r represents a state reachable in r time steps. By definition, $P_t^r(Y_r)$ represents an over-approximation of the states reachable in r time steps. Hence, $P_t^r(Y_r)$ holds for any assignment to the variables in Y_r representing a state reachable in r time steps. Thus, $\textsc{Bmc}_j^k(M) \to P_t^r(Y_r)$, with $r \leq k$. By definition, $P_t^r(Y_r)$ is usable for $\textsc{Bmc}_j^k(M)$, with $r \leq k$. Observe that there is no upper bound on the value of t.

(ii) Observe that $P_t^r(Y_{k-t})$ represents an under-approximation of the states which do not satisfy the failing property in t time steps. Hence, $P_t^r(Y_{k-t}) \to \neg\textsc{Bmc}_j^k(M)$ with $t \leq k$. Consequently, $\textsc{Bmc}_j^k(M) \to \neg P_t^r(Y_{k-t})$. By definition, $P_t^r(Y_{k-t})$ is usable for $\textsc{Bmc}_j^k(M)$, with $t \leq k$. Observe that there is no upper bound on the value of r.

□

Theorem 4.4 *Let $\textsc{Bmc}_j^k(M)$ be given by (9). Then the following holds:*

(i) *$Q_t^r(Y_{k-t})$ is usable for $\textsc{Bmc}_j^k(M)$, with $r \geq 0$ and $0 \leq t \leq k$.*

(ii) *$\neg Q_t^r(Y_r)$ is usable for $\textsc{Bmc}_j^k(M)$, with $t \geq 0$ and $0 \leq r \leq k$.*

Proof. The proof is similar to the proof for Theorem 4.3.

(i) If $\textsc{Bmc}_j^k(M)$ is satisfiable, then $\textsc{Tran}_{k-t}^k(M) \wedge \textsc{Prop}_k^k(M)$, with $t \leq k$ is also satisfiable and Y_{k-t} represents a state that satisfies the failing property in t time steps. By definition, $Q_t^r(Y_{k-t})$ represents an over-approximation of the states that satisfy the failing property in t time steps. Thus, $\textsc{Bmc}_j^k(M) \to Q_t^r(Y_{k-t})$, with $t \leq k$. By definition, $Q_t^r(Y_{k-t})$ is usable for $\textsc{Bmc}_j^k(M)$, with $t \leq k$. Observe that there is no upper bound on the value of r.

(ii) Observe that $Q_t^r(Y_r)$ represents an under-approximation of the states that are unreachable r in time steps. Hence, $Q_t^r(Y_r) \to \neg\textsc{Bmc}_j^k(M)$, with $r \leq k$. Consequently, $\textsc{Bmc}_j^k(M) \to \neg Q_t^r(Y_r)$. By definition, $Q_t^r(Y_r)$ is usable for $\textsc{Bmc}_j^k(M)$, with $r \leq k$. Observe that there is no upper bound on the value of t.

□

Remark 4.5 Even though we describe the most general setting for learning and reusing interpolants, the specific interpolants computed in the standard interpolant-based fixed point condition [21] are also usable according to the conditions of Theorems 4.3 and 4.4. Hence, interpolant reuse can be readily integrated in a standard interpolant-based UMC flow.

Remark 4.6 The conditions of Theorems 4.3 and 4.4 can be used in *any* BMC/UMC setting, independently of whether a fixed point is used and whether it is based on interpolants.

Instance	w/o interpolants	w/ interpolants
6-bit counter	1.51	5.29
7-bit counter	16.38	61.03
8-bit counter	236.90	784.81
I1	7.08	7.11
I2	31.36	36.96
I3	38.36	60.60
I4	52.45	58.25
I5	150.54	157.81

Table 1
Results with and without interpolant reuse

Remark 4.7 It is straightforward to conclude that reverse interpolants can be used for developing a fixed point condition alternative to the one of [21]. The advantages of this alternative fixed point condition are expected to depend on the actual automaton.

5 Experimental Results

The practical experience reported in this section respects a preliminary SAT-based model checking prototype. The prototype represents interpolants as Reduced Boolean Circuits (RBCs) [1]. The backend SAT solver is MiniSAT [11]. The implementation of interpolant computation is still preliminary and, currently, different interpolants do not share structure. Even though each interpolant is generated with the rules of [1], each different interpolant is maintained with a separate RBC manager, and so common nodes among different interpolants are not shared. Moreover, the utilization of interpolants was evaluated in a standard BMC loop, and so interpolants were solely computed for search pruning purposes. Iinterpolants were computed with respect to the last time step and reused in the last time step. As a result, reused interpolants serve for preventing sets of unwanted states to be reached.

Table 1 shows preliminary results from interpolant reuse. The first set of instances represent standard counters, for which counterexample exists. The second set of instances represent industrial problem instances, for which a counterexample also exists. As can be concluded, the utilization of interpolants does not yield improvements to the run times. For the first set of (artificial) examples the results are worse than for the second set of (industrial) examples. As mentioned above, the setup for the utilization of interpolants is certainly not the most adequate. We considered a simple BMC loop, where interpolants are solely used for search pruning purposes. The reuse of interpolants in a UMC setting is expected to provide more competitive results, since the interpolants have be computed for checking the fixed point condition.

6 Conclusions and Future Work

This paper develops conditions for learning and reusing of interpolants in SAT-based model checking. Computed interpolants can be used for requiring states from a set of states or for preventing states from a set of states.

The preliminary results are not positive, albeit the implementation is still very preliminary. Moreover, the experimental setup chosen was not beneficial for the reuse of interpolants. Instead of an interpolant-based UMC algorithm, where interpolants need to be computed, our experiments consisted of a standard BMC loop, where computed interpolants were solely used for search pruning purposes.

A few drawbacks of the current implementation have been identified. Examples include the lack of structure sharing between different interpolants, and the fact that interpolants were computed solely for interpolant reuse and not for checking the existence of a fixed point. A future implementation of these improvements is expected to yield more promising results from interpolant reuse.

References

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT solvers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2000.

[2] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Advances in Computers*, chapter Bounded Model Checking. Academic Press, 2003.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, March 1999.

[4] P. Bjesse and K. Claesen. SAT-based verification without state space traversal. In *International Conference on Formal Methods in Computer-Aided Design*, 2000.

[5] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *International Conference on Formal Methods in Computer-Aided Design*, 2002.

[6] E. M. Clarke, O. Grumberg, and A. Peled. *Model Checking.* MIT Press, 1999.

[7] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *International Conference on Computer-Aided Verification*, 2001.

[8] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, July 1962.

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, July 1960.

[11] N. Een and N. Sorensson. An extensible SAT solver. In *Sixth International Conference on Theory and Applications of Satisfiability Testing*, May 2003.

[12] N. Een and N. Sorensson. Temporal induction by incremental SAT solving. In *Workshop on Bounded Model Checking*, volume 89 of *ENTCS*, 2003.

[13] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Vardi. Multiple-counterexample guided iterative abstraction refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 2003.

[14] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Design, Automation and Test in Europe Conference*, pages 142–149, March 2002.

[15] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *National Conference on Artificial Intelligence*, pages 431–437, July 1998.

[16] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *International Conference on Computer-Aided Design*, November 2003.

[17] H.-J. Kang and I.-C. Park. SAT-based unbounded symbolic model checking. In *Design Automation Conference*, pages 840–843, June 2003.

[18] J. P. Marques-Silva. Improvements to the implementation of interpolant-based model checking. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, October 2005.

[19] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[20] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *International Conference on Computer-Aided Verification*, July 2002.

[21] K. L. McMillan. Interpolation and SAT-based model checking. In *International Conference on Computer-Aided Verification*, 2003.

[22] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 2003.

[23] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, June 2001.

[24] P. Pudlák. Lower bounds for resolution and cutting planes proofs and monotone circuit computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.

[25] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT solver. In *International Conference on Formal Methods in Computer-Aided Design*, 2000.

[26] O. Strichman. Tuning SAT checkers for bounded model checking. In *International Conference on Computer-Aided Verification*, July 2000.

[27] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968.

[28] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe Conference*, pages 10880–10885, March 2003.

A compact linear translation for bounded model checking [1]

Paul B. Jackson [2]

*School of Informatics, University of Edinburgh,
Kings Buildings, Edinburgh EH9 3JZ, United Kingdom*

Daniel Sheridan [3]

*Adelard LLP, 10 Northampton Square,
London EC1V 0HB, United Kingdom*

Abstract

We present a syntactic scheme for translating future-time LTL bounded model checking problems into propositional satisfiability problems. The scheme is similar in principle to the *Separated Normal Form* encoding proposed in [5] and extended to past time in [3]: an initial phase involves putting LTL formulae into a normal form based on linear-time fixpoint characterisations of temporal operators.

As with [3] and [7], the size of propositional formulae produced is linear in the model checking bound, but the constant of proportionality appears to be lower.

A denotational approach is taken in the presentation which is significantly more rigorous than that in [5] and [3], and which provides an elegant alternative way of viewing fixpoint based translations in [7] and [1].

Key words: Bounded Model Checking, Linear Temporal Logic, Fixpoints, SAT, Denotational Semantics

1 Introduction

Frisch, Sheridan and Walsh [5] proposed a scheme for translating LTL bounded model checking problems into satisfiability problems that is significantly different from the original bounded model checking encoding scheme presented in [2]. This scheme involves simplifying temporal formulae using rules based on fixpoint characterisations of temporal operators to put formulae into a

[1] This research was funded in part by UK EPSRC Grant GR/N64243/01
[2] Email: pbj@inf.ed.ac.uk
[3] Email: dan.sheridan@contact.org.uk

separated normal form (SNF) similar to that used by Fisher in his temporal resolution work [4]. Frisch, Sheridan and Walsh [5] showed that this new scheme had significant advantages in terms of compactness of propositional formulae generated and SAT solver run times. This SNF approach smoothly extends to handle past time LTL [3] and has similarities with automata-based translations [8].

In this paper, we present an alternate set of simplification rules for future time LTL that again exploits fixpoint characterisations, but is simpler to describe. As with [3] and [7], the size of propositional formulae produced is linear in the model checking bound,[4] and the constant of proportionality is smaller than with [7].

A major contribution of the paper is in providing a denotational semantics approach to justifying the encoding. This justification is much more complete and rigorous than that in [5] and [3], and it enables easy exploration of variations on these and other encodings.

A minor novelty is that we experiment with using an abstract symbolic representation of Kripke structures. Most formal presentations of BMC conflate a description of the BMC translation from LTL syntax to propositional logic syntax with a description of its semantics, and only informally refer to possible symbolic representations (for example, using propositional formulae, BDDs or Boolean circuits) of Kripke structures. Our approach allows us to keep the translation and semantics distinct. While our approach is more verbose, we argue that it is easier to understand, especially when handling the auxiliary variables introduced by our translation.

Our implementation is not yet complete so we do not have empirical data on SAT solver performance on the resulting encoded problems. We certainly expect the performance to be no worse than with SNF because of the similarity.

The structure of the rest of the paper is as follows. In Section 2 we present the foundations of our denotational approach, closely following the logic of the original BMC translation from [2]. Section 3 then gives a high-level overview of our new translation. The translation is split into two phases: the normalisation phase is covered in Section 4 and the translation to propositional logic phase in Section 5. Section 6 covers related work and we draw our conclusions in Section 7.

2 Preliminaries

2.1 Syntax for LTL

Fix some set V of Boolean-valued state variables. We use these as the atomic propositions of our LTL formulae. We initially consider LTL formulae de-

[4] The super-linear behaviour of the SNF encoding as noted in [7] was obtained with an older version of the SNF code than that presented in [3]

scribed by the grammar

$$\phi ::= v \mid \neg v \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathbf{X}\,\phi \mid \mathbf{F}\,\phi \mid \mathbf{G}\,\phi \mid \phi\,\mathbf{U}\,\phi \mid \phi\,\mathbf{R}\,\phi$$

where $v \in V$. Such formulae are in *negation normal form* (NNF): negations are only applied to state variables. Any LTL formula can be transformed into an equivalent NNF formula by pushing negations inwards. We use $\phi \Rightarrow \psi$ as an abbreviation for $\neg\phi \vee \psi$.

2.2 Kripke structures

The set of states S associated with a set of state variables V is the set of valuations $V \to \mathbb{B}$ of those variables. A *Kripke structure* M over a set of state variables V is a pair $\langle I, T\rangle$ where $I \subseteq S$ is a set of initial states and $T \subseteq S \times S$ is a transition relation which has to be total. Many treatments of Kripke Structures consider the set of states S more abstractly and introduce a labelling function specifying which atomic propositions are true in each state. It is straightforward to adapt our presentation to this more general approach, but, for simplicity, we do not. A similar simplification is common in automata-based approaches to LTL model checking. An *unconstrained path* π over M is an infinite sequence of states $\pi = s_0, s_1, \ldots$ where $s_i \in S$. A *constrained path* or simply a *path* over M must satisfy the constraints $s_0 \in I$ and, for every $i \geq 0$, $\langle s_i, s_{i+1}\rangle \in T$. Let $\mathsf{Paths}(M)$ be the set of all paths over M. A *finite path* is a finite prefix of a path. A finite path $s_0, s_1, \ldots, s_{k-1}$ *has bound* k.

We denote distinct copies of the set of state variables using superscripts. For example, V', V^i for $i \in \mathbb{N}$. If $v \in V$, the corresponding variable in V' is v' and in V^i is v^i. A *symbolic Kripke structure* $\hat{M}$ over a set of state variables V is a pair $\langle \hat{I}, \hat{T}\rangle$ where $\hat{I}(V)$ is a symbolic representation of the set of initial states and $\hat{T}(V, V')$ is a symbolic representation of the transition relation. The notation $A(V)$ here indicates that the symbolic representation A is over the variables V. We then write elsewhere $A(W)$ for A with the variables V replaced with the variables W. The Kripke structure corresponding to $\hat{M}$ has $I \doteq \{s \in S \mid s \models \hat{I}\}$ and $T \doteq \{\langle s, t\rangle \in S \times S \mid s, t \models \hat{T}\}$. This definition uses satisfiability relations $\models$ for single states and pairs of states satisfying a propositional formula defined in the expected way.

2.3 Infinite path semantics

A common approach to LTL semantics is to define an inductive relation $\pi \models^i \phi$ indicating at which positions $i \in \mathbb{N}$ on path π the LTL formula ϕ is satisfied. We give an exactly equivalent definition in a denotational style. We define the infinite denotation ${}^{\pi}[\![\phi]\!]$ of formula ϕ to be an infinite sequence of $a_0, a_1, \ldots$ of Boolean values, elements of $\mathbb{B} = \{\bot, \top\}$, such that a_i is true just when ϕ is satisfied at position i of path π. We write the set of all such infinite boolean sequences as $\mathbb{B}^\omega$. We often view a sequence $a \in \mathbb{B}^\omega$ as a function of type

$\mathbb{N} \to \mathbb{B}$, and refer to element i as $a(i)$. When we say that a formula is satisfied by a path without indicating an explicit position on the path, we mean that the formula is satisfied at position 0. Formally, the *infinite denotation* of an LTL formula is given inductively by:

$$\begin{array}{llll} {}^{\pi}[\![v]\!](i) = s_i(v) & {}^{\pi}[\![\mathbf{O}\,\phi]\!] & = [\![\mathbf{O}]\!]({}^{\pi}[\![\phi]\!]) & \text{for } \mathbf{O} \in \{\mathbf{X}, \mathbf{F}, \mathbf{G}\} \\ {}^{\pi}[\![\neg v]\!] \;\; = [\![\neg]\!]({}^{\pi}[\![v]\!]) & {}^{\pi}[\![\phi\,\mathbf{O}\,\psi]\!] & = [\![\mathbf{O}]\!]({}^{\pi}[\![\phi]\!], {}^{\pi}[\![\psi]\!]) & \text{for } \mathbf{O} \in \{\wedge, \vee, \mathbf{U}, \mathbf{R}\} \end{array}$$

where the individual operator denotations are given by

$$\begin{array}{lcllcl} [\![\neg]\!](a)(i) & \doteq & \neg a(i) & [\![\mathbf{F}]\!](a)(i) & \doteq & \exists j \geq i.\ a(j) \\ [\![\wedge]\!](a,b)(i) & \doteq & a(i) \wedge b(i) & [\![\mathbf{G}]\!](a)(i) & \doteq & \forall j \geq i.\ a(j) \\ [\![\vee]\!](a,b)(i) & \doteq & a(i) \vee b(i) & [\![\mathbf{U}]\!](a,b)(i) & \doteq & \exists j \geq i.\ b(j) \wedge \forall n \in \{i \mathinner{..} j-1\}.\ a(n) \\ [\![\mathbf{X}]\!](a)(i) & \doteq & a(i+1) & [\![\mathbf{R}]\!](a,b)(i) & \doteq & \forall j \geq i.\ b(j) \vee \exists n \in \{i \mathinner{..} j-1\}.\ a(n) \end{array}$$

Here $a, b \in \mathbb{B}^{\omega}$ are infinite denotations and $i \in \mathbb{N}$ indexes positions in denotations. These explicit denotations for operators help simplify the presentation later. Their use emphasises that the meaning of operators is dependent only on the meaning of subformulae, not on the syntactic structure of subformulae.

Let us write $\phi \equiv \psi$ when LTL formulae ϕ and ψ have the same infinite denotation for all Kripke structures M and paths π over those structures.

2.4 Finite denotations when paths are looping

In producing finite propositional encodings of model checking problems, bounded model checking works with finite representations of infinite paths and infinite denotations. In this subsection we consider *loop case* representations. In the next subsection we consider *prefix case* representations.

In the loop case with bound k and loop start l where $0 \leq l < k$, a finite path $\dot{\pi} = s_0, \ldots, s_{k-1}$ such that $T(s_{k-1}, s_l)$ represents the infinite path $s_0 \ldots s_{l-1}(s_l \ldots s_{k-1})^{\omega}$. We call such infinite paths *(k, l) loop paths*. Similarly, finite loop-case denotations such as $\dot{a} = a_0, \ldots, a_{k-1}$ where $a_i \in \mathbb{B}$ represent infinite denotations $a_0 \ldots a_{l-1}(a_l \ldots a_{k-1})^{\omega}$. A *loop-case inflation* function $\uparrow_{\circ}^{\infty}$ maps finite paths and denotations to the corresponding infinite paths and denotations. A *restriction* function $|_k$ maps (k, l) loop paths and infinite loop-case denotations to their finite representations.

When working with loop paths and their finite denotations, we can define a *finite loop-case denotation function* ${}^{\dot{\pi}}_{l}\overset{\mathrm{F}}{[\![\phi]\!]}_k$ with range $\mathbb{B}^k$ that exactly mimics the infinite denotation function:

$$^{\dot{\pi}\uparrow_{\circ}^{\infty}}[\![\phi]\!] \;=\; {}^{\dot{\pi}}_{l}\overset{\mathrm{F}}{[\![\phi]\!]}_k \uparrow_{\circ}^{\infty}$$

where $\dot{\pi}$ is a k-bounded path representing a (k, l) loop path. The definition is similar to that of the infinite denotation with the following changes:

$$
{}_l\overset{\mathrm{F}}{[\![\mathbf{X}]\!]}_k(\dot{a})(i) \doteq \begin{cases} \dot{a}(i{+}1) & \text{if } i < k{-}1 \\ \dot{a}(l) & \text{if } i = k{-}1 \end{cases}
\qquad
\begin{aligned}
{}_l\overset{\mathrm{F}}{[\![\mathbf{F}]\!]}_k(\dot{a})(i) &\doteq \exists j \in \{\min(i,l) \mathinner{..} k{-}1\}.\ \dot{a}(j) \\
{}_l\overset{\mathrm{F}}{[\![\mathbf{G}]\!]}_k(\dot{a})(i) &\doteq \forall j \in \{\min(i,l) \mathinner{..} k{-}1\}.\ \dot{a}(j)
\end{aligned}
$$

$$
\begin{aligned}
{}_l\overset{\mathrm{F}}{[\![\mathbf{U}]\!]}_k(\dot{a}, \dot{b})(i) &\doteq (\exists j \in \{i \mathinner{..} k{-}1\}.\ \dot{b}(j) \wedge \forall n \in \{i \mathinner{..} j{-}1\}.\ \dot{a}(n)) \\
&\quad \vee \exists j \in \{l \mathinner{..} i{-}1\}.\ \dot{b}(j) \wedge \forall n \in \{i \mathinner{..} k{-}1\} \cup \{l \mathinner{..} j{-}1\}.\ \dot{a}(n) \\
{}_l\overset{\mathrm{F}}{[\![\mathbf{R}]\!]}_k(\dot{a}, \dot{b})(i) &\doteq (\forall j \in \{i \mathinner{..} k{-}1\}.\ \dot{b}(j) \vee \exists n \in \{i \mathinner{..} j{-}1\}.\ \dot{a}(n)) \\
&\quad \wedge \forall j \in \{l \mathinner{..} i{-}1\}.\ \dot{b}(j) \vee \exists n \in \{i \mathinner{..} k{-}1\} \cup \{l \mathinner{..} j{-}1\}.\ \dot{a}(n)
\end{aligned}
$$

where $\dot{a}, \dot{b} \in \mathbb{B}^k$ are finite denotations and index i is in range $\{0 \mathinner{..} k{-}1\}$.

All quantifications in the finite loop-case denotation function are over finite ranges. Following the denotation function's structure, we can define an executable *loop-case translation function* ${}_l[\phi]_k^i$ that can translate a question concerning the existence of a finite path loop-case satisfying a formula into a propositional satisfiability question. The relationship between the loop-case denotation and translation functions is expressed by:

$$
{}^{\dot{\pi}}_{\,l}\overset{\mathrm{F}}{[\![\phi]\!]}_k(i) \quad \Leftrightarrow \quad \dot{\pi} \models {}_l[\phi]_k^i
$$

where the relation $\dot{\pi} \models q$ for when a finite path $\dot{\pi}$ satisfies a propositional formula q is defined in the expected way. Representative cases of the definition of the loop-case translation function are

$$
{}_l[v]_k^i \doteq v^i \qquad\qquad {}_l[\mathbf{F}\,\phi]_k^i \doteq \bigvee_{j=\min(i,l)}^{k-1} {}_l[\phi]_k^j
$$

In the original paper introducing BMC [2] and many other papers in the BMC literature, symbolic Kripke structures are not explicitly introduced and confusing semantic notations occur in translation function definitions. For example, the base case of the translation function might be written as ${}_l[v]_k^i \doteq v(s_i)$ where a state variable v is treated as function which it is not, and a state s_i is introduced which is part of the semantic presentation, not part of the language of propositional logic that is being translated into.

2.5 Finite denotations when paths have common prefix

In the prefix case with bound k, a finite path $\dot{\pi} = s_0, \ldots, s_{k-1}$ represents the set of all paths that have it as a prefix. Prefix-case denotations such as $\dot{a} = a_0, \ldots, a_{k-1}$ where $a_i \in \mathbb{B}$ represent infinite denotations $\dot{a}\bot^\omega$. A *prefix-case inflation* function $\uparrow^\infty$ maps finite denotations to the corresponding infinite denotations. The restriction function $\pi|_k$ introduced in the last section is also used to select the k-bounded prefix of an infinite path π.

We define a *finite prefix-case denotation function* $\overset{\text{F}}{{}^{\dot\pi}[\![\phi]\!]_k}$ (or sometimes $\overset{\text{F}}{{}^{\dot\pi}_{-}[\![\phi]\!]_k}$) in a similar way to the the finite loop-case denotation function. In this case, the denotations for the LTL temporal operators are given by:

$$\overset{\text{F}}{[\![\mathbf{X}]\!]}_k(\dot a)(i) \doteq \begin{cases} \dot a(i+1) & \text{if } i < k-1 \\ \bot & \text{if } i = k-1 \end{cases} \qquad \begin{aligned} \overset{\text{F}}{[\![\mathbf{F}]\!]}_k(\dot a)(i) &\doteq \exists j \in \{i \mathinner{..} k-1\}.\ \dot a(j) \\ \overset{\text{F}}{[\![\mathbf{G}]\!]}_k(\dot a)(i) &\doteq \bot \end{aligned}$$

$$\overset{\text{F}}{[\![\mathbf{U}]\!]}_k(\dot a, \dot b)(i) \doteq \exists j \in \{i \mathinner{..} k-1\}.\ \dot b(j) \wedge \forall n \in \{i \mathinner{..} j-1\}.\ \dot a(n)$$

$$\overset{\text{F}}{[\![\mathbf{R}]\!]}_k(\dot a, \dot b)(i) \doteq \exists j \in \{i \mathinner{..} k-1\}.\ \dot a(j) \wedge \forall n \in \{i \mathinner{..} j\}.\ \dot b(n)$$

The prefix case denotation underapproximates the standard infinite denotation and so is sound. We can express this by the assertion

$$\overset{\text{F}}{{}^{\pi|_k}[\![\phi]\!]_k} \uparrow^\infty \sqsubseteq {}^{\pi}[\![\phi]\!]$$

where π is any infinite path and we are treating the domain of infinite denotations $\mathbb{B}^\omega$ as a lattice with order relation $a \sqsubseteq b \doteq \forall i \in \mathbb{N}.\ a(i) \Rightarrow b(i)$. As with the loop-case, we can derive a prefix translation function $[\phi]_k^i$ (sometimes written as ${}_{-}[\phi]_k^i$) from the prefix-case denotation function.

3 The new full translation

We describe here the high-level structure and properties of our translation in order to motivate the details in subsequent sections. The translation takes an LTL formula ϕ, symbolic Kripke structure $\hat M$ and bound k and creates a propositional formula that is satisfiable just when some path in $\hat M$ with a k-bounded representation satisfies ϕ. Conceptually the translation proceeds in 3 stages:

1. Apply normalisation function $\mathcal{N}()$ to ϕ to create normalised temporal logic formula ψ.
2. Create a formula
$$[\hat M]_k \wedge \left([\psi]_k^0 \vee \bigvee_{l=0}^{k-1} {}_lL_k(\hat M) \wedge {}_l[\psi]_k^0\right) \tag{1}$$
that brings together the prefix case and loop case translations of ψ and correspondingly checks that an unconstrained finite path is representing a prefix case or a loop case path. Here $[\hat M]_k \doteq \hat I(V^0) \wedge \bigwedge_{i=0}^{k-2} \hat T(V^i, V^{i+1})$ generates a proposition for checking that a finite state sequence is a finite path and ${}_lL_k(\hat M) \doteq \hat T(V^{k-1}, V^l)$ is a constraint for specifying that a finite path represents a (k, l) loop path.

3. Apply standard logic transformations so as to collect together common factors in the disjuncts of Formula (1) and ensure the formula's size is linear in k.

The original translation of [2] consists of step 2 without steps 1 and 3. Even with careful optimisations, the size of the original translation is claimed in [7] to be cubic in the worst case. Our normalisation in step 1 enables the factoring for linear size in step 3.

We formally write our full translation as:

$$\mathrm{Full}[\hat{M}, \phi]_k \doteq \mathrm{body}\big(\mathrm{Norm}[\ \hat{M},\ \mathcal{N}(\phi)\ ]_k\big)$$

where the normalised-formula translation function $\mathrm{Norm}[\hat{M}, \psi]_k$ groups together steps 2 and 3. This translation function produces formulae of form $\exists\, \overline{z}.\ q$ where $\overline{z}$ is a vector of propositional variables and q is a propositional logic formula. The function body() returns the body q of such formulae. This existential quantification $\exists \overline{z}$ arises because $\mathcal{N}()$ produces formulae in LTL extended with existential quantification. See Section 4 for a full definition of $\mathcal{N}()$ and Section 5 for a full definition of $\mathrm{Norm}[\hat{M}, \psi]_k$.

To state the correctness of our full translation, we introduce a reference semantics which combines the infinite and finite prefix-case semantics. A (k, l) loop path π *satisfies at bound* k an LTL formula ϕ if π satisfies it in the standard infinite semantics (if ${}^{\pi}[\![\phi]\!](0)$ holds). If π is not a (k, l) loop path for any l, then π *satisfies at bound* k a formula ϕ if the k-bounded prefix of π satisfies ϕ in the finite prefix case semantics (if ${}^{\pi|_k}\overset{\mathrm{F}}{[\![\phi]\!]}_k(0)$ holds). A formula ϕ is *existentially valid with bound* k in Kripke structure M, written $M \models_k \mathbf{E}\,\phi$, when some path π of M satifies ϕ at bound k. We can now state the overall correctness claim for our new translation as follows.

Theorem 3.1 ***(Correctness of new translation).*** *For any symbolic Kripke structure $\hat{M}$ with corresponding semantic structure M, any LTL formula ϕ and any bound $k > 0$, we have that*

$$M \models_k \mathbf{E}\,\phi \quad \Leftrightarrow \quad \mathrm{Full}[\hat{M}, \phi]_k \textit{ is satisfiable}$$

4 Formula normalisation

4.1 Overview

Normalisation proceeds in two main stages. Firstly the LTL operators **F**, **G**, **U**and **R** in the input formula are all converted into forms involving greatest fixpoint operators. Section 4.3 handles how this is done with **G** and **R**, operators with natural greatest fixpoint characterisations and Section 4.4 handles the more subtle case of **F** and **U** which have natural least fixpoint characterisations. Secondly, as described in Section 4.5, each greatest fixpoint expression is converted into a form involving existential quantification at the outermost level of the formula. Section 4.5 also explains why least fixpoint characterisations cannot be handled. Normalisation also involves some renaming transforms on **X** in the input formula and on certain new formulae produced in the first normalisation stage. Section 4.6 covers renaming transforms in general. Finally Section 4.7 gives a self-contained summary of the normalisation function.

The interesting part of the proof of Theorem 3.1 involves showing the following two equations concerning normalisation:

$$ {}^{\dot\pi\uparrow^{\infty}_{\circ}}\llbracket\phi\rrbracket = {}^{\dot\pi}_{l}\overset{\mathrm{F}}{\llbracket}\mathcal{N}(\phi)\rrbracket_k \uparrow^{\infty}_{\circ} \tag{2}$$

$$ {}^{\dot\pi}\overset{\mathrm{F}}{\llbracket}\phi\rrbracket_k = {}^{\dot\pi}\overset{\mathrm{F}}{\llbracket}\mathcal{N}(\phi)\rrbracket_k \tag{3}$$

where ϕ is any LTL formula and $l \in \{0\mathinner{..}k-1\}$. Equation (2) states that the finite loop-case denotation of normalized formulae is equivalent to their standard infinite denotation. Equation (3) states that the prefix-case denotation is preserved by normalisation. The subsections which follow include assertions of equalities which are intermediate steps in the proofs of Equation (2) and Equation (3).

We write $\phi \equiv_L \psi$ ($\phi \equiv_P \psi$) when two formulae always have the same finite loop-case (prefix-case) denotation, and $\phi \equiv_F \psi$ when they always have the same denotation under both finite semantics.

4.2 Extending LTL with a greatest fixpoint operator

We add to the syntax of LTL formulae *timed variables* α (also known as *flexible variables*), and greatest fixpoint expressions $\nu\alpha.\ \phi$ with infinite and finite semantics:

$$\begin{array}{llll}
{}^{\pi}\llbracket\alpha\rrbracket^{\rho} & = \rho(\alpha) & {}^{\pi}_{l}\overset{\mathrm{F}}{\llbracket}\alpha\rrbracket^{\dot\rho}_k & = \dot\rho(\alpha) \\
{}^{\pi}\llbracket\lambda\alpha.\phi\rrbracket^{\rho} & = \lambda a \in \mathbb{B}^{\omega}.\ {}^{\pi}\llbracket\phi\rrbracket^{\rho[\alpha\mapsto a]} & {}^{\dot\pi}_{l}\overset{\mathrm{F}}{\llbracket}\lambda\alpha.\phi\rrbracket^{\dot\rho}_k & = \lambda\dot a \in \mathbb{B}^{k}.\ {}^{\dot\pi}_{l}\overset{\mathrm{F}}{\llbracket}\phi\rrbracket^{\dot\rho[\alpha\mapsto\dot a]}_k \\
{}^{\pi}\llbracket\nu\alpha.\phi\rrbracket^{\rho} & = \mathrm{gfp}\big({}^{\pi}\llbracket\lambda\alpha.\phi\rrbracket^{\rho}\big) & {}^{\dot\pi}_{l}\overset{\mathrm{F}}{\llbracket}\nu\alpha.\phi\rrbracket^{\dot\rho}_k & = \mathrm{gfp}\big({}^{\dot\pi}_{l}\overset{\mathrm{F}}{\llbracket}\lambda\alpha.\phi\rrbracket^{\dot\rho}_k\big)
\end{array}$$

where $l \in \{0\mathinner{..}k-1\} \cup \{-\}$. Lambda abstractions $\lambda\alpha.\phi$ are examples of *unary function formulae*. To provide meaning in the semantics for free variables, we extend the semantic functions with an environment argument ρ or $\dot\rho$. An *unbounded environment* ρ maps each free variable to an infinite sequence in $\mathbb{B}^{\omega}$ and a *k-bounded environment* $\dot\rho$ maps each free variable to a finite sequence in $\mathbb{B}^{k}$. In the other previously-defined clauses of the semantic functions, the environments are recursively propagated down unchanged.

The greatest fixpoint operator gfp is given the standard definition from the Tarski-Knaster construction. Let F be a monotone function of type $D \to D$ on a complete lattice $\langle D, \sqsubseteq\rangle$ with least upper bound operator $\bigsqcup$. We have that $\mathrm{gfp}(F) \ \doteq\ \bigsqcup\{x \in D | x \sqsubseteq F(x)\}$. In all our semantics D is of form $R \to \mathbb{B}$. In the infinite semantics $R = \mathbb{N}$ and in both finite semantics $R = \{0\mathinner{..}k-1\}$. The order relation $\sqsubseteq$ and least upper bound operation $\bigsqcup$ are defined pointwise:

$x \sqsubseteq y \doteq \forall i.\ x(i) \Rightarrow y(i)$ and $(\bigsqcup S)(i) \doteq \exists x \in S.\ x(i)$ where lattice elements $x, y \in R \to \mathbb{B}$, set of elements $S \subseteq R \to \mathbb{B}$ and index $i \in R$.

4.3 Greatest fixpoint characterisations for ***G*** *and* ***R***

Fixpoint versions of the *globally* operator $\mathbf{G}$ and the *release* operator $\mathbf{R}$ are

$$\tilde{\mathbf{G}}\,\beta \doteq \nu\alpha.\ \beta \wedge \mathbf{X}\,\alpha \qquad\qquad \beta\,\tilde{\mathbf{R}}\,\gamma \doteq \nu\alpha.\ \gamma \wedge (\beta \vee \mathbf{X}\,\alpha)$$

Our following discussion focusses the $\tilde{\mathbf{G}}$ operator. It extends very straightforwardly to cover the $\tilde{\mathbf{R}}$ operator too.

It is well known that the standard $\mathbf{G}$ is equivalent to this fixpoint version in the infinite semantics: $\mathbf{G}\,\beta \equiv \tilde{\mathbf{G}}\,\beta$. It is straightforward to check that this equivalence also holds in the prefix semantics. For example, it is easy to show $\mathbf{G}\,\beta \equiv_P \tilde{\mathbf{G}}\,\beta$ once one observes that $\lambda\alpha.\ \beta \wedge \mathbf{X}\,\alpha$ has a unique fixpoint in the prefix semantics when a binding for β is fixed. Indeed, if one adds least fixpoint operators $\mu\alpha.\ \phi$ to LTL, one can also make the definitions

$$\tilde{\mathbf{F}}\,\beta \doteq \mu\alpha.\ \beta \vee \mathbf{X}\,\alpha \qquad\qquad \beta\,\tilde{\mathbf{U}}\,\gamma \doteq \mu\alpha.\ \gamma \vee (\beta \wedge \mathbf{X}\,\alpha)$$

and show $\mathbf{F}\,\beta \equiv_P \tilde{\mathbf{F}}\,\beta$ and $\beta\,\mathbf{U}\,\gamma \equiv_P \beta\,\tilde{\mathbf{U}}\,\gamma$. This provides some justification for the naturalness of the prefix-case semantics of the LTL operators.

In the proof of Equation (2), an appropriate stage of normalisation for shifting to the finite semantics is after $\tilde{\mathbf{G}}$ and $\tilde{\mathbf{R}}$ have been introduced. With $l \in \{0 \mathinner{..} k-1\}$ and $\dot{b} \in \mathbb{B}^k$, we have the following:

$$[\![\tilde{\mathbf{G}}]\!](\dot{b}\uparrow^{\infty}_{\circ}) = \big({}_{l}\overset{\mathrm{F}}{[\![}\tilde{\mathbf{G}}]\!]_k(\dot{b})\big)\uparrow^{\infty}_{\circ}$$

4.4 Greatest fixpoint characterisations for ***F*** *and* ***U***

As noted in the previous section, in the prefix case the fixpoint with operators $\tilde{\mathbf{F}}$ and $\tilde{\mathbf{U}}$ is unique, the least and greatest fixpoints are the same. For example, we have that: $\tilde{\mathbf{F}}\,\beta \equiv_P \nu\alpha.\ \beta \vee \mathbf{X}\,\alpha$. The loop-case is not so simple. Consider the loop-case semantics for $\mathbf{F}$.

$${}_{l}\overset{\mathrm{F}}{[\![}\mathbf{F}]\!]_k(\dot{a})(i) = \exists j \in \{\min(i,l) \mathinner{..} k-1\}.\ \dot{a}(j)$$

The right-hand side here is equivalent to

$$(\exists j \in \{i \mathinner{..} k-1\}.\ \dot{a}(j)) \ \vee\ (\exists j \in \{l \mathinner{..} k-1\}.\ \dot{a}(j))$$

Each disjunct here is an instance of the prefix semantics for $\mathbf{F}$ and, as above, we know we can switch to greatest fixpoints in the prefix semantics. We craft some definitions of new operators to take advantage of this observation.

Let us introduce variations $\mathbf{X}^{\top}$ and $\mathbf{X}^{\perp}$ on the next step operator that have non-looping semantics even in the loop case. Their finite semantics is

$$ {}_l\overset{\text{F}}{[\![}\mathbf{X}^\top]\!]_k(\dot a)(i) \doteq \begin{cases}\dot a(i+1) & \text{if } i < k-1\\ \top & \text{if } i = k-1\end{cases} \quad {}_l\overset{\text{F}}{[\![}\mathbf{X}^\perp]\!]_k(\dot a)(i) \doteq \begin{cases}\dot a(i+1) & \text{if } i < k-1\\ \perp & \text{if } i = k-1\end{cases} $$

where $l \in \{0 \mathinner{..} k-1\} \cup \{-\}$. We use these in the definitions

$$ \begin{aligned} \tilde{\mathbf{F}}^\perp\, \beta &\doteq \nu\alpha.\ \beta \vee \mathbf{X}^\perp\, \alpha & \qquad \tilde{\mathbf{G}}^\top\, \beta \doteq \nu\alpha.\ \beta \wedge \mathbf{X}^\top\, \alpha \\ \beta\, \tilde{\mathbf{U}}^\perp\, \gamma &\doteq \nu\alpha.\ \gamma \vee (\beta \wedge \mathbf{X}^\perp\, \alpha) \end{aligned} $$

These newly introduced fixpoint operators have the following semantic characterisations in both the prefix and the loop cases.

$$ \begin{aligned} {}_l\overset{\text{F}}{[\![}\tilde{\mathbf{F}}^\perp]\!]_k(\dot a)(i) &= \exists j \in \{i \mathinner{..} k-1\}.\ \dot a(j) \\ {}_l\overset{\text{F}}{[\![}\tilde{\mathbf{G}}^\top]\!]_k(\dot a)(i) &= \forall j \in \{i \mathinner{..} k-1\}.\ \dot a(j) \\ {}_l\overset{\text{F}}{[\![}\tilde{\mathbf{U}}^\perp]\!]_k(\dot a, \dot b)(i) &= \exists j \in \{i \mathinner{..} k-1\}.\ \dot b(j) \wedge \forall n \in \{i \mathinner{..} j-1\}.\ \dot a(n) \end{aligned} $$

To force consideration of the semantics of an operator at the loop start in the loop case, we introduce a unary LTL operator **loopstart** with semantics

$$ {}_l\overset{\text{F}}{[\![}\mathbf{loopstart}]\!]_k(\dot a)(i) \doteq \begin{cases}\dot a(l) & \text{if } l \in \{0 \mathinner{..} k-1\}\\ \perp & \text{if } l = -\end{cases} $$

With these new operators at hand, we have the following identities allowing us to replace **F** and **U** with expressions involving greatest fixpoint operators.

$$ \begin{aligned} \mathbf{F}\,\alpha &\equiv_F \tilde{\mathbf{F}}^\perp\,\alpha \ \vee\ \mathbf{loopstart}\,\tilde{\mathbf{F}}^\perp\,\alpha \\ \alpha\,\mathbf{U}\,\beta &\equiv_F \alpha\,\tilde{\mathbf{U}}^\perp\,\beta \ \vee\ (\tilde{\mathbf{G}}^\top\,\alpha \ \wedge\ \mathbf{loopstart}(\alpha\,\tilde{\mathbf{U}}^\perp\,\beta)) \end{aligned} $$

The identity involving **F** can readily be derived using facts presented above. The main steps are:

$$ \begin{aligned} &{}^{\pi}_{l}\overset{\text{F}}{[\![}\tilde{\mathbf{F}}^\perp\,\alpha \ \vee\ \mathbf{loopstart}\,\tilde{\mathbf{F}}^\perp\,\alpha]\!]_k(i) \ =\ {}^{\pi}_{l}\overset{\text{F}}{[\![}\tilde{\mathbf{F}}^\perp\,\alpha]\!]_k(i) \ \vee\ {}^{\pi}_{l}\overset{\text{F}}{[\![}\tilde{\mathbf{F}}^\perp\,\alpha]\!]_k(l) \\ &= (\exists j \in \{i \mathinner{..} k-1\}.\ {}^{\pi}_{l}\overset{\text{F}}{[\![}\alpha]\!]_k(j)) \ \vee\ (\exists j \in \{l \mathinner{..} k-1\}.\ {}^{\pi}_{l}\overset{\text{F}}{[\![}\alpha]\!]_k(j)) \ =\ {}^{\pi}_{l}\overset{\text{F}}{[\![}\mathbf{F}\,\alpha]\!]_k(i) \end{aligned} $$

These identities are also closely related to those discussed in Section 6.1.

4.5 Expressing greatest fixpoints using existential operators

We focus on the cases of the two finite semantics since these are the cases we need. A similar discussion applies with the infinite semantics.

Let us augment our LTL syntax with existential quantification over timed variables $\exists\alpha.\ \phi$ and a *globally from the start* operator $\mathbf{G}_0$ which have finite semantics

$$
\begin{aligned}
{}^{\dot\pi}_{l}\overset{\mathrm{F}}{[\![}\exists\alpha.\ \phi]\!]^{\dot\rho}_{k}(i) &\doteq \exists \dot a \in \mathbb{B}^k.\ {}^{\dot\pi}_{l}\overset{\mathrm{F}}{[\![}\phi]\!]^{\dot\rho[\alpha\mapsto\dot a]}_{k}(i)\\
{}_{l}\overset{\mathrm{F}}{[\![}\mathbf{G}_0]\!]_{k}(\dot a)(i) &\doteq \forall j \in \{0\,..\,k-1\}.\ \dot a(j)
\end{aligned}
$$

where $0 \le i < k$, $\dot a \in \mathbb{B}^k$ and $l \in \{0..k-1\} \cup \{-\}$. Note that the globally-from-the-start operator $\mathbf{G}_0$ always quantifies over the full time range, no matter what index i we consider its value at, even in the prefix case.

Using these definitions we can phrase an identity for eliminating greatest fixpoint expressions occurring in contexts, buried under other operators:

$$\Psi[\nu\alpha.\ \phi] \equiv_F \exists\alpha.\ \mathbf{G}_0\,(\alpha \Rightarrow \phi) \wedge \Psi[\alpha]$$

where context expression Ψ is a unary function formula with monotone denotation, and the notation $\cdot[\cdot]$ is the application operator for such functions.

The existential quantification derives from the least-upper-bound operator in the definition of the gfp operator, and semantics of the formula $\mathbf{G}_0\,(\alpha \Rightarrow \phi)$ captures the $x \sqsubseteq F(x)$ constraint in the definition body (see Section 4.2).

The corresponding identity for an lfp (least-fixpoint) operator involves a universal quantification derived from the greatest-lower-bound operator in the lfp operator definition. Since our goal is to eventually produce satisfiability problems, we cannot make use of this identity.

4.6 Renamings

An LTL formula in some context is *renamed* if it replaced by a new timed variable which is asserted equivalent to it. When contexts are monotone, it is sufficient to assert an implicational relationship between the new variable and the renamed formula. We have that

$$\Psi[\phi] \ \equiv_F\ \exists\alpha.\ \mathbf{G}_0\,(\alpha \Rightarrow \phi) \wedge \Psi[\alpha]$$

where Ψ is a monotone unary function formula.

In some cases, the formula to be replaced is *time invariant*: it has denotation $\bot^k$ or $\top^k$. In these cases, it is sufficient to replace it by an *untimed variable* (sometimes called a rigid variable) and use existential quantification over untimed variables. Let us add untimed variables x to the LTL syntax and existential quantification over them $\exists x.\ \phi$ with semantics:

$${}^{\dot\pi}_{l}\overset{\mathrm{F}}{[\![}x]\!]^{\dot\rho}_{k}(i) \ =\ \dot\rho(x) \qquad\qquad {}^{\dot\pi}_{l}\overset{\mathrm{F}}{[\![}\exists x.\ \phi]\!]^{\dot\rho}_{k}(i) \ =\ \exists a_0 \in \mathbb{B}.\ {}^{\dot\pi}_{l}\overset{\mathrm{F}}{[\![}\phi]\!]^{\dot\rho[x\mapsto a_0]}_{k}(i)$$

where $0 \le i < k$, $\dot a \in \mathbb{B}^k$ and $l \in \{0\,..\,k-1\} \cup \{-\}$, and we extend the notion of environment $\dot\rho$ to provide Boolean-valued bindings for untimed variables. We then have:

$$\Psi[\phi] \ \equiv_F\ \exists x.\ (x \Rightarrow \phi) \wedge \Psi[x]$$

where Ψ is a monotone unary function formula and ϕ a time invariant formula.

4.7 The normalisation function

We assemble here the results from the previous subsections into a single overall definition of the normalisation function $\mathcal{N}()$. Assume that formulae to start are in negation normal form. $\mathcal{N}()$ applies the following transformation rules:

$$\begin{array}{ll}
\Psi[\mathbf{G}\, f] & \longrightarrow \exists\alpha.\ \Psi[\alpha] \ \wedge\ \mathbf{G}_0\,(\alpha \ \Rightarrow\ f \wedge \mathbf{X}\,\alpha) \\[1em]
\Psi[f\,\mathbf{R}\, g] & \longrightarrow \exists\alpha.\ \Psi[\alpha] \ \wedge\ \mathbf{G}_0\,(\alpha \ \Rightarrow\ g \wedge (f \vee \mathbf{X}\,\alpha)) \\[1em]
\Psi[\mathbf{X}\, f] & \longrightarrow \exists\alpha.\ \Psi[\alpha] \ \wedge\ \mathbf{G}_0\,(\alpha \ \Rightarrow\ \mathbf{X}\, f) \\[1em]
\Psi[\mathbf{F}\, f] & \longrightarrow \exists\alpha, x.\ \Psi[\alpha \vee x] \ \wedge\ \mathbf{G}_0\,(\alpha \ \Rightarrow\ f \vee \mathbf{X}^{\perp}\,\alpha) \wedge (x \Rightarrow \mathbf{loopstart}\,\alpha) \\[1em]
\Psi[f\,\mathbf{U}\, g] & \longrightarrow \exists\alpha, \beta, x.\ \ \Psi[\alpha \vee (\beta \wedge x)] \ \wedge\ \mathbf{G}_0\,(\alpha \ \Rightarrow\ g \vee (f \wedge \mathbf{X}^{\perp}\,\alpha)) \\
& \qquad\qquad\qquad \wedge\, \mathbf{G}_0\,(\beta \ \Rightarrow\ f \wedge \mathbf{X}^{\top}\,\beta) \ \wedge\ (x \ \Rightarrow\ \mathbf{loopstart}\,\alpha)
\end{array}$$

These rules are applied in a single bottom-up pass over the initial formula. To suggest this bottom-up direction, the subformulae f and g are required to be propositional, free from temporal operators. Rules are not applied to any of the new generated structure, for example, new $\mathbf{X}$s. Usual assumptions are made about variables bound by the existential quantifiers being suitably renamed to avoid any unintentional capture of variables. An example of applying the normalisation function is

$$\begin{array}{lll}
\mathbf{F}\,\mathbf{G}\,\neg p & \longrightarrow\ \exists\alpha.\ \underline{\mathbf{F}\,\alpha}\ \wedge\ \mathbf{G}_0\,(\alpha \Rightarrow \neg p\ \wedge\ \mathbf{X}\,\alpha) & \textit{by } \mathbf{G} \textit{ rule} \\[0.5em]
& \longrightarrow\ \exists\alpha, \beta, x.\ \underline{(\beta \vee x)}\ \wedge \mathbf{G}_0\,(\beta \Rightarrow \alpha \vee \mathbf{X}^{\perp}\,\beta) & \\
& \qquad\qquad \wedge\ (x \Rightarrow \mathbf{loopstart}\,\beta)\ \wedge\ \mathbf{G}_0\,(\alpha \Rightarrow \neg p \wedge \mathbf{X}\,\alpha) & \textit{by } \mathbf{F} \textit{ rule}
\end{array}$$

where, in the intermediate expression, we have underlined the partially reduced input formula that is about to be transformed by a second rule, and, in the final expression, the propositional residue of the input formula.

The resulting formulae have normal form

$$\exists\overline{\alpha},\, \overline{x}.\ R\ \wedge\ LS\ \wedge\ \mathbf{G}_0\,(X\ \wedge\ X^{*})$$

where $\overline{\alpha}$ is a vector of timed variables, $\overline{x}$ is a vector of untimed variables, R is the residual top-level propositional structure of the initial formula, LS is a conjunction of formulae of form $x \Rightarrow \mathbf{loopstart}\,\alpha$, X is a conjunction of formulae of form $\alpha \Rightarrow f[\mathbf{X}\,g]$ where context f and formula g are propositional, and X^{*} is a conjunction of formulae of form $\alpha \Rightarrow f[\mathbf{X}^{\top}\,g]$ and $\alpha \Rightarrow f[\mathbf{X}^{\perp}\,g]$ where again context f and formula g are propositional.

The function $\mathcal{N}(\phi)$ can be computed in time linear in the size ϕ.

5 Translation of normalised formulae

The loop-case and prefix-case translation functions over the syntax of the components R, LS, X and X^* of our normalised formulae are as follows:

$$
\begin{array}{lll}
{}_l[\alpha]_k^i = \alpha_i & {}_l[v]_k^i \; = v^i & {}_l[\phi \wedge \psi]_k^i = {}_l[\phi]_k^i \wedge {}_l[\psi]_k^i \\[1em]
{}_l[x]_k^i = x & {}_l[\neg\phi]_k^i = \neg {}_l[\phi]_k^i & {}_l[\phi \vee \psi]_k^i = {}_l[\phi]_k^i \vee {}_l[\psi]_k^i
\end{array}
$$

$$
{}_l[\mathbf{X}\,\phi]_k^i = \begin{cases} {}_l[\phi]_k^{i+1} & \text{if } i < k{-}1 \\ \bot & \text{if } i = k{-}1 \\ & \text{and } l = - \\ {}_l[\phi]_k^l & \text{if } i = k{-}1 \\ & \text{and } l \in \{0 \,..\, k{-}1\} \end{cases}
\qquad
{}_l[\mathbf{X}^\top\,\phi]_k^i = \begin{cases} {}_l[\phi]_k^{i+1} & \text{if } i < k{-}1 \\ \top & \text{if } i = k{-}1 \end{cases}
$$

$$
{}_l[\textbf{loopstart}\,\phi]_k^i = \begin{cases} \bot & \text{if } l = - \\ {}_l[\phi]_k^l & \text{if } l \in \{0 \,..\, k{-}1\} \end{cases}
\qquad
{}_l[\mathbf{X}^\bot\,\phi]_k^i = \begin{cases} {}_l[\phi]_k^{i+1} & \text{if } i < k{-}1 \\ \bot & \text{if } i = k{-}1 \end{cases}
$$

where $l \in \{0 \,..\, k{-}1\}$ for the loop case and $l = -$ for the prefix case. The translation function for formulae ψ in the normal form described at the end of the last section is:

$$
\mathrm{Norm}[\hat{M}, \psi]_k \doteq \exists \overline{z}.\ [\hat{M}]_k \;\wedge\; [R]_k^0 \;\wedge\; \textstyle\bigwedge_{i=0}^{k-2} [X]_k^i \;\wedge\; \bigwedge_{i=0}^{k-1} [X^*]_k^i \;\wedge
$$
$$
\big(\, ([LS]_k^0 \;\wedge\; [X]_k^{k-1}) \;\vee\; \textstyle\bigvee_{l=0}^{k-1} ({}_lL_k(\hat{M}) \;\wedge\; {}_l[LS]_k^0 \;\wedge\; {}_l[X]_k^{k-1})\,\big)
$$

where $[\hat{M}]_k$ and ${}_lL_k(\hat{M})$ are as defined in Section 3 and the vector of propositional variables $\overline{z}$ contains variables $\alpha_0, \ldots, \alpha_{k-1}$ for each timed variable α in $\overline{\alpha}$ and a variable x for each untimed variable x in $\overline{x}$. The resulting formula has size linear in k, $|\phi|$ and $|\hat{M}|$. More precisely, its size is $O(|\hat{I}| + k \cdot (|\phi| + |\hat{T}|))$.

6 Related work

6.1 Helsinki work

The BMC translations closest to ours are those of [7] and [6]. These translations are also linear in k and they exploit fixpoint characterisations of operators. A core observation in [7] from the viewpoint of this paper is that the loop-case denotations of the LTL operators **F**, **G**, **U** and **R** are all equivalent to the restriction to bound k of the denotation of non-looping versions of the operators at bound $k + (k - l) - 1$. For example:

$$
{}_l\overset{\mathrm{F}}{[\![\mathbf{G}]\!]}_k(\dot{a}) \;=\; \Big({}_l\overset{\mathrm{F}}{[\![\tilde{\mathbf{G}}^\top]\!]}_k(\dot{a}{\uparrow}_\circ^{k+(k-l)-1})\Big)|_k
$$

where $\dot{a} \in \mathbb{B}^k$, $l \in \{0 \mathinner{..} k{-}1\}$, $\dot{a} \uparrow_{\circ}^{k'} \doteq \dot{a} \uparrow_{\circ}^{\infty}|_{k'}$ unrolls a loop denotation to bound k' and $\tilde{\mathbf{G}}^{\top}$ is as defined in Section 4.4. The justification in [7] for these identities is rather indirect and involves appealing to arguments about fixpoints in CTL. However, we note that we can prove them straightforwardly using some of the same insights as are necessary to prove the identity

$$^{\dot{\pi}\uparrow_{\circ}^{\infty}}[\![\phi]\!] \;=\; {}_{l}^{\dot{\pi}}\overset{\mathrm{F}}{[\![\phi]\!]}_{k} \uparrow_{\circ}^{\infty}$$

introduced in Section 2.4 which is at the heart of the justification of the original bounded model checking translation of [2].

A major apparent difference is that the approach in [7] introduces very few auxiliary variables by encoding to reduced Boolean circuits (RBCs), a DAG representation of Boolean formulae. However, when these circuits are subsequently translated into CNF, auxiliary variables are introduced for many of the internal nodes of the circuits, and we guess that one gets roughly one new auxiliary variable per fixpoint step, the same as what we use.

Comparing the sizes of resulting propositional formulae in the approach of [7] to ours, we observe that our encoding for **G** and **R** involves unrolling the fixpoint functions for k rather than $2k$ steps, and so involves introducing about half the number of $\wedge$s and $\vee$s. For **F** and **U** the number of $\wedge$s and $\vee$s introduced appears to be more similar, though for **F** we introduce roughly half the number of auxiliary variables into the final CNF formulae.

The approach of more recent work [6] from the same group is more similar to an automata-based approach in that the fixpoint constraints on auxiliary variables in the loop case also have a loop shape. Ignoring the incremental and past-time aspects of [6], the numbers of operators and auxiliary variables introduced seem to be slightly closer to those with our approach.

Experimentation and more detailed analysis are needed to sharpen the above preliminary remarks and importantly to compare how the approaches affect SAT run times.

6.2 Other work

The BMC journal paper [1] gives a translation exploiting fixpoint characterisations, though the encoding size is not linear in the bound. As written, the translation is not sound: it appears to be using a greatest fixpoint characterisation for *all* the LTL operators which is clearly unsound for **F** and **U**. We speculate that this mistake could have been avoided if the translation had been derived within a formal framework such as presented in this paper.

We observe that a recent NuSMV release (V2.3.1, Nov 2005) seems to use a similar translation that is sound. This translation has some similarities to that of [7] discussed above in Section 6.1 in that the loop case translation is calculated using non-looping fixpoint constraints.

7 Conclusions

We have presented a translation for future time LTL bounded model checking that is linear in the bound k and more compact than competing translations, in particular that of [7].

We have also presented a rigorous framework for analysing translations. Both the body of the paper and the discussion of related work show the usefulness of the framework, and it is expected that it will be of significant use in exploring future variations on and extensions to BMC translations.

References

[1] Biere, A., A. Cimatti, E. M. Clarke, O. Strichman and Y. Zhu, *Bounded model checking*, Advances in Computers **58** (2003).

[2] Biere, A., A. Cimatti, E. M. Clarke and Y. Zhu, *Symbolic model checking without BDDs*, in: W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS 99*, Lecture Notes in Computer Science **1579** (1999), pp. 193–207.

[3] Cimatti, A., M. Roveri and D. Sheridan, *Bounded verification of past LTL*, in: A. J. Hu and A. K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer Aided Design (FMCAD 2004)*, Lecture Notes in Computer Science (2004).

[4] Fisher, M., *A resolution method for temporal logic*, in: *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI)* (1991).

[5] Frisch, A., D. Sheridan and T. Walsh, *A fixpoint based encoding for bounded model checking*, in: M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design; 4th International Conference, FMCAD 2002*, Lecture Notes in Computer Science **2517** (2002), pp. 238–254.

[6] Heljanko, K., T. Junttila and T. Latvala, *Incremental and complete bounded model checkinfg for full pltl*, in: K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification: 17th International Conference, CAV 2005*, Lecture Notes in Computer Science **3576** (2005), pp. 98–111.

[7] Latvala, T., A. Biere, K. Heljanko and T. Junttila, *Simple bounded LTL model checking*, in: *Formal Methods in Computer-Aided Design; 5th International Conference, FMCAD 2004*, Lecture Notes in Computer Science **3312** (2004), pp. 186–200.

[8] Sheridan, D., *Bounded model checking with SNF, alternating automata and Büchi automata*, in: *Second International Workshop on Bounded Model Checking*, Electronic Notes in Theoretical Computer Science (2004).

Authors