Encodings of Bounded LTL Model Checking in Effectively Propositional Logic

Juan Antonio Navarro-Pérez and Andrei Voronkov

The University of Manchester
School of Computer Science
{navarroj,voronkov}@cs.manchester.ac.uk

Abstract. We present an encoding that is able to specify LTL bounded model checking problems within the Bernays-Schönfinkel fragment of first-order logic. This fragment, which also corresponds to the category of effectively propositional problems (EPR) of the CASC system competitions, allows a natural and succinct representation of both a software/hardware system and the property that one wants to verify.
The encoding for the transition system produces a formula whose size is linear with respect to its original description in common component description languages used in the field (e.g. smv format) preserving its modularity and hierarchical structure. Likewise, the LTL property is encoded in a formula of linear size with respect to the input formula, plus an additional component, with a size of $O(\log k)$ where k is the bound, that represents the execution flow of the system.
The encoding of bounded model checking problems by effectively propositional formulae is the main contribution of this paper. As a side effect, we obtain a rich collection of benchmarks with close links to real-life applications for the automated reasoning community.

1 Introduction

Model checking is a technique suitable for verifying that a hardware or software component works according to some formally specified expected behaviour. This is usually done by building a description of the system, often modelled as a finite state machine in a formal language suitable for further deployment, and using a temporal logic to specify properties that the system is expected to satisfy.

One of the first advances in model checking consists of the use of symbolic model checkers [9], where the transition system of the finite state machine is represented symbolically. These symbolic representations, which usually take the form of a binary decision diagram (BDD), provide significant improvements over previous techniques; but some formulae are still hard to encode succinctly using BDDs and, moreover, the encoding itself is often highly sensitive to the variable order used to create the representation.

Another significant achievement in the state of the art of model checking came when Biere et al. [1] proposed a technique now widely known as bounded model checking (BMC). In bounded model checking instead of trying to prove

the correctness of the given property, one searches for counterexamples within executions of the system of a bounded length. A propositional formula is created and a decision procedure for propositional logic, such as DPLL [5], is used to find models which in turn represent bugs in the system. When no models are found the bound is increased trying to search for longer counterexamples.

Although the basic method is not complete by itself, i.e. it can only disprove properties, it has been found as an useful tool to quickly find simple bugs in systems [1, 4, 11] and a good complement to other BDD based techniques. A significant amount of research has been spent recently on extending this technique to more expressive temporal logics [6], obtaining better propositional encodings [7], and proposing termination checks to regain completeness [10]. A recent survey on the state of the art is found in the work of Biere et al. [2].

Bounded model checking has been largely focused on generating and solving problems encoded in propositional logic. We observe, however, that BMC problems can also be easily and naturally encoded within the Bernays-Schönfinkel class of formulae. One of our motivations is to obtain a new source of problems for first-order reasoners, particularly those geared towards the effectively propositional division (EPR) of the CASC system competitions [12]. Problems in this category are non-propositional but with a finite Herbrand Universe and lie within the Bernays-Schönfinkel fragment.

Moreover, we believe that the EPR encoding has several advantages over the propositional approach. First, it gives a more succinct and natural description of both the system and the property to verify. It is not needed, for example, to replicate copies of the temporal formula for every step of the execution trace where it has to be checked. Furthermore, it is possible to directly translate systems descriptions written in a modular, without requiring to flatten or expand module definitions before the encoding. A prover could potentially use this information to better organise the search for a proof or counterexample.

On the other hand, our encoding may also turn out to be useful for propositional, SAT-based, approaches to bounded model checking. Indeed, it preserves the structure of the original bounded model checking problem in the obtained effectively propositional formula and reduces the problem of finding an optimised propositional encoding to the problem of finding an optimised propositional instantiation of the EPR description.

After introducing a number of formal definitions in Section 2, we present in Section 3 two different encodings of linear temporal logic (LTL) into effectively propositional formulae. The first encoding takes an LTL formula and a bound k, and produces a set of constraints that captures the execution paths satisfying the temporal property. The second encoding is an improvement that produces two sets of constraints: one that depends on the LTL formula only (i.e. not the bound) and its output is linear with respect to its input; and another, with a size of $O(k)$, that depends on the bound k only. Compare with propositional encodings where, if n is the size of the LTL formula, the output is typically of size $O(nk)$ instead of $O(n + k)$ with our approach. Furthermore, with a binary encoding of states, the size of the later component can be reduced to $O(\log^2 k)$.

We also present, in Section 4, an approach to the encoding of modular descriptions of model checking problems so as to preserve their modularity and hierarchical representation. We show in particular how several features of a software/hardware description language such as smv [3] can be easily represented within the effectively propositional fragment. Using the ideas depicted here, it is also possible to develop a tool to automatically translate system descriptions in industry standard formats (e.g. smv or verilog) into a format such as tptp [13] suitable for consumption by first order theorem provers.

2 Background

In this section we introduce the main formal definitions that are used throughout this paper. We first define the linear temporal logic (LTL) in a way that closely follows the standard definitions found in literature but with a few modifications to better represent the notion of bounded executions. changed!

Definition 1. Let $\mathcal{V} = \{p_1, \ldots, p_n\}$ be a set of elements called *state variables*. A subset $s \subseteq \mathcal{V}$ is known as a *state*.

A *path* $\pi = s_0 s_1 \ldots$ is a, finite or infinite, sequence of states. The *length* of a finite path $\pi = s_0 \ldots s_k$, denoted by $|\pi|$, is $k+1$; while, for an infinite path, we define $|\pi| = \omega$, where $\omega > k$ for every number k.

A *k-path* is either a finite path of the form $\pi = s_0 \ldots s_k$, or an infinite path with a loop of the form $\pi = s_0 \ldots s_{l-1} s_l \ldots s_k s_l \ldots s_k \ldots$, in the sequel also written as $\pi = s_0 \ldots s_{l-1}(s_l \ldots s_k)^\omega$. □

We will assume that system executions are always infinite paths, i.e. there are no deadlock states. Finite paths, however, are also needed to represent the prefix of an execution of the system up to a bounded length. With this intuition in mind we now define the semantics of LTL formulae in negation normal form; these are formulae built using propositional and temporal connectives, but negation is only allowed in front of atomic propositions. changed!

Definition 2. A path $\pi = s_0, s_1, \ldots$ is a *model* of an LTL formula ϕ at a state s_i, where $i < |\pi|$, denoted by $\pi \models_i \phi$, if

$$
\begin{array}{lll}
\pi \models_i p & \text{iff} & p \in s_i, \\
\pi \models_i \neg p & \text{iff} & p \notin s_i, \\
\pi \models_i \psi \wedge \phi & \text{iff} & \pi \models_i \psi \text{ and } \pi \models_i \phi, \\
\pi \models_i \psi \vee \phi & \text{iff} & \pi \models_i \psi \text{ or } \pi \models_i \phi, \\
\pi \models_i \mathbf{X}\phi & \text{iff} & i+1 < |\pi| \text{ and } \pi \models_{i+1} \phi, \\
\pi \models_i \mathbf{F}\phi & \text{iff} & \exists j,\ i \leq j < |\pi|,\ \pi \models_j \phi, \\
\pi \models_i \psi \mathbf{W} \phi & \text{iff} & \text{either: } \pi \text{ is infinite and } \forall j,\ i \leq j,\ \pi \models_j \psi, \\
 & & \text{or: } \exists j',\ i \leq j' < |\pi|,\ \pi \models_{j'} \phi \text{ and } \forall j,\ i \leq j < j',\ \pi \models_j \psi.
\end{array}
$$

Also π is a model of $\top$ for every state s_i with $i < |\pi|$, and of $\bot$ for no state. We write $\pi \models \phi$ to denote $\pi \models_0 \phi$. □

Note that we introduced the *weak until*, $\mathbf{W}$, as a primary connective of our temporal logic. Other standard temporal connectives —such as *until*, *release* and *globally*— can be introduced as abbreviations of the other existing connectives: $\psi \mathbf{U} \phi = \mathbf{F}\phi \wedge (\psi \mathbf{W} \phi)$, $\psi \mathbf{R} \phi = \phi \mathbf{W} (\psi \wedge \phi)$, and $\mathbf{G}\phi = \phi \mathbf{W} \bot$.

changed! If we consider infinite paths only, then the definition given matches the standard notion of LTL that can be found in literature; in particular dualities such as $\neg \mathbf{F}\phi \equiv \mathbf{G}\neg\phi$ do hold. Since we assume that system executions are always infinite, one can make use of these identities to put formulae into negation normal form without any loss of generality.

changed! Now the finite case is defined so that if $\pi \models_i \phi$ then, for all possible infinite paths π' extending π, it is also the case that $\pi' \models_i \phi$. Here we deviate a little from usual definitions of LTL and dualities such as the above-mentioned do not hold anymore. For example, neither $\mathbf{F}\phi$ nor $\mathbf{G}\neg\phi$ hold in a finite path where $\neg\phi$ holds at all states. In particular, since finite paths in the temporal logic defined are interpreted as prefixes of longer paths, one can not write a formula to test for the end of a path.

changed! **Definition 3.** A *Kripke structure* over a set of state variables $\mathcal{V}$ is a tuple $M = (S, I, T)$ where $S = 2^{\mathcal{V}}$ is the set of all states, $I \subseteq S$ is a set whose elements are called *initial states*, and T is a binary relation on states, $T \subseteq S \times S$, called the *transition relation* of the system. We also make the assumption that the transition relation is total, i.e. for every state $s \in S$ there is a state $s' \in S$ such that $(s, s') \in T$.

changed! A *path* $\pi = s_0 s_1 \ldots$ *is in the structure* M if $s_0 \in I$ and for every $0 < i < |\pi|$ we have $(s_{i-1}, s_i) \in T$. We say that a path π in M is a *prefix path* if it is finite, and a *proper path* otherwise.

An LTL formula ϕ is *satisfiable* in a Kripke structure M if there is a proper path π in M such that $\pi \models \phi$. Similarly, a formula ϕ is *valid* in M if, for every proper path π in M, $\pi \models \phi$. □

changed! Note that, if π is a prefix path in M and $\pi \models \phi$, then for every extension π' of π we also have $\pi' \models \phi$, thus prefix paths are enough for testing satisfiability. Observe, however, that formulae such as $\mathbf{G}\psi$ or $\psi \mathbf{W} \phi$ (where ϕ never holds) are never satisfied by (finite) prefix paths.

We proceed now to formally introduce the fragment of quantifier free predicate logic which is the target language of our main translation. This fragment, also known as the Bernays-Schönfinkel class of formulae, does not allow the use of function symbols or arbitrary quantification. Only variables and constant symbols are allowed as terms, and variables are assumed to be universally quantified. We also define its semantics using Herbrand interpretations.

Definition 4. We assume given a set of *predicate symbols* $\mathcal{P}$, a finite set of *constant symbols* $\mathcal{D} = \{\mathsf{s}_0, \ldots, \mathsf{s}_k\}$, and a set of *variables* which we will usually denote by uppercase letters: X, Y, ... The set $\mathcal{D}$ is sometimes referred to as the *domain* of the logic. A *term* is either a variable or a constant symbol. An *atom* is an expression of the form $\mathsf{p}(t_1, \ldots, t_n)$ where $\mathsf{p} \in \mathcal{P}$ and each t_i is a term. A *ground atom* is an atom all whose terms are constant symbols.

Quantifier-free predicate formulae are built from atoms using the standard propositional connectives ($\top$, $\bot$, $\wedge$, $\vee$, $\neg$). Other connectives can be introduced as abbreviations: $F \rightarrow G \equiv \neg F \vee G$. A *ground formula* is a formula built using only ground atoms. A *ground instance* of a formula F is any ground formula obtained by uniformly replacing the variables in F with constant symbols.

A *Herbrand interpretation* is a set of ground atoms. The notion of whether a Herbrand interpretation $\mathcal{I}$ is a *model* of a ground formula F, denoted by $\mathcal{I} \models F$, is defined in the usual way:

$$\begin{array}{lll} \mathcal{I} \models A & \text{iff} & A \in \mathcal{I}, \\ \mathcal{I} \models \neg F & \text{iff} & \mathcal{I} \not\models F, \end{array} \qquad \begin{array}{lll} \mathcal{I} \models F \wedge G & \text{iff} & \mathcal{I} \models F \text{ and } \mathcal{I} \models G, \\ \mathcal{I} \models F \vee G & \text{iff} & \mathcal{I} \models F \text{ or } \mathcal{I} \models G. \end{array}$$

Also $\mathcal{I}$ is always a model of $\top$ and never of $\bot$. Now a Herbrand interpretation $\mathcal{I}$ is said to be a *model of a non-ground formula* F if it is a model of every ground instance of F, and a model of a set of formulae if it is a model of every formula in the set. A set of formulae, also referred to as a *set of constraints*, is called *satisfiable* if it has at least one model. □

Since we will only be dealing with quantifier-free formulae and Herbrand interpretations, we will often simply say *predicate formula* when we refer to a quantifier-free predicate formula and *interpretation* when we refer to a Herbrand interpretation. Also note that, while the symbol s_i represents a state in a path, s_i represents a constant symbol in a predicate formulae. The similar notation was chosen intentionally since a constant s_i will be used as a symbolic representation of a state s_i. The intended meaning should always be clear by context, but a different typeface is also used as a hint to distinguish the two possibilities.

Similarly, it is assumed throughout this paper that $\mathcal{P}$ contains a unary predicate symbol p for every state variable $p \in \mathcal{V}$. The atom $\mathsf{p}(\mathsf{s}_i)$ symbolically represents the fact that a variable p is true at the state s_i of a path (i.e. $p \in s_i$), and the symbol $\mathcal{P}_\mathcal{V}$ denotes the set of predicates representing state variables. Our next aim is to define a notion of symbolic representation of Kripke structures along the lines of representations commonly used in the propositional case.

Let us define the *canonical first-order structure* for $\mathcal{P}_\mathcal{V}$, denoted by $C_\mathcal{V}$. This structure is an interpretation which, instead of the symbolic representations s_i used elsewhere, draws constant symbols from the domain $2^\mathcal{V}$, its signature the set of predicate symbols $\mathcal{P}_\mathcal{V}$, and the interpretation of every predicate $\mathsf{p} \in \mathcal{P}_\mathcal{V}$ is defined as $C_\mathcal{V} \models \mathsf{p}(s)$ iff $p \in s$. changed!

Definition 5. Let $I(X)$ and $T(X, Y)$ be predicate formulae of variables X and X, Y, respectively, using predicate symbols $\mathcal{P}_\mathcal{V}$ and no constants. We say that this pair of formulae *symbolically represents* a Kripke structure M if

1. a state s is an initial state of M iff $C_\mathcal{V} \models I(s)$.
2. a pair (s, s') belongs to the transition relation of M iff $C_\mathcal{V} \models T(s, s')$. □

The idea used in this definition extends to represent paths in a Kripke structure M by Herbrand interpretations as follows.

Definition 6. Given an interpretation $\mathcal{I}$ over the domain $\mathcal{D} = \{\mathsf{s}_0, \ldots, \mathsf{s}_k\}$, we define *the k-path induced by* $\mathcal{I}$, denoted by $\pi^{\mathcal{I}}$, by $\pi^{\mathcal{I}} = s_0^{\mathcal{I}} \ldots s_k^{\mathcal{I}}$, where $s_i^{\mathcal{I}} = \{p \in \mathcal{V} \mid \mathcal{I} \models \mathsf{p}(\mathsf{s}_i)\}$, for all $0 \leq i \leq k$. We will rather informally refer to the states $s_i^{\mathcal{I}}$ as *induced states*. For the induced k-path $\pi^{\mathcal{I}}$ we will often omit the superscripts on the induced states and simply write $\pi^{\mathcal{I}} = s_0 \ldots s_k$.

Given a value l with $0 \leq l \leq k$, we also introduce the notation $\pi^{l,\mathcal{I}}$ to denote the infinite k-path $s_0 \ldots s_{l-1}(s_l \ldots s_k)^\omega$ with a loop starting at s_l. □

In the sequel we will assume that the set of initial states I and the transition relation T of our Kripke structures are always symbolically described in this way. Also, we will normally consider only interpretations $\mathcal{I}$ over the domain $\mathcal{D} = \{\mathsf{s}_0, \ldots, \mathsf{s}_k\}$. Then $s_i^{\mathcal{I}}$ means the induced state along the k-path $s_0^{\mathcal{I}}, \ldots, s_k^{\mathcal{I}}$ induced by $\mathcal{I}$. Definition 6 immediately implies the following fact.

Lemma 1. *Let $M = (S, I, T)$ be a Kripke structure and $\mathcal{I}$ an interpretation.*

1. *$\mathcal{I} \models I(\mathsf{s}_i)$ iff $s_i^{\mathcal{I}}$ is an initial state of M.*
2. *$\mathcal{I} \models T(\mathsf{s}_i, \mathsf{s}_j)$ iff $(s_i^{\mathcal{I}}, s_j^{\mathcal{I}})$ belongs to the transition relation of M.* □

3 Encoding of temporal properties

In this section we present a translation that allows one to encode an LTL formula as a quantifier free predicate formula. Following the results from Biere et al. [1], it has been shown that, if one wants to check the satisfiability of an LTL formula, it is enough to search for k-paths that satisfy this formula.

Theorem 1 (Biere et al. [1]). *An LTL formula ϕ is satisfiable in a Kripke structure M iff, for some k, there is a k-path π in M with $\pi \models \phi$.*

Our translation makes use of this result by creating, for a given value k and a Kripke structure M, a predicate formula whose models correspond to k-paths of the system satisfying the original LTL formula (the details of such correspondence are given later in Proposition 2). We begin giving a set of constraints that characterise the k-paths of Kripke structures and define some auxiliary symbols, which are used later in the translation.

Definition 7. Let $M = (S, I, T)$ be a Kripke structure, and also let $k \geq 0$. The *predicate encoding of k-paths*, denoted by $[\![\, k \,]\!]$, is defined as the set of constraints:

$$
\begin{array}{l}
\mathsf{succ}(\mathsf{s}_0, \mathsf{s}_1) \\
\mathsf{succ}(\mathsf{s}_1, \mathsf{s}_2) \\
\ldots \\
\mathsf{succ}(\mathsf{s}_{k-1}, \mathsf{s}_k) \\
\mathsf{succ}(X, Y) \rightarrow \mathsf{less}(X, Y) \\
\mathsf{succ}(X, Y) \wedge \mathsf{less}(Y, Z) \rightarrow \mathsf{less}(X, Z) \\
\mathsf{succ}(X, Y) \rightarrow \mathsf{trans}(X, Y) \\
\mathsf{hasloop} \rightarrow \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_0) \vee \cdots \vee \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_k)
\end{array}
$$

And the *predicate encoding* of the structure M, denoted by $|[\, M \,]|$, is defined as:

$$\mathsf{trans}(X, Y) \rightarrow T(X, Y)$$
$$I(\mathsf{s}_0)$$

We also define $|[\, M, k \,]| = |[\, M \,]| \cup |[\, k \,]|$. Note that of the predicates $\mathsf{succ}(X, Y)$, $\mathsf{less}(X, Y)$, $\mathsf{trans}(X, Y)$ and $\mathsf{hasloop}$ are fresh new predicates not in $\mathcal{P}_\mathcal{V}$. □

The intuition behind the predicates introduced in the previous definition is to model paths in the Kripke structure. It easily follows, for example, that if an interpretation $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_i, \mathsf{s}_j)$ then the pair $(s_i^\mathcal{I}, s_j^\mathcal{I})$ is in the transition relation of the structure. The encoding of temporal formulae can then use the $\mathsf{hasloop}$ predicate as a trigger to enforce paths accepted as models to be infinite, since it would make $\mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$ true for some l. The following proposition summarises important properties of the models of $|[\, M, k \,]|$. changed!

Proposition 1. *Let M be a Kripke structure, and let $\mathcal{I}$ be a model of the set of constraints $|[\, M, k \,]|$. Then for every $0 \leq i, j, l \leq k$:*

1. *If $i < j$ then $\mathcal{I} \models \mathsf{less}(\mathsf{s}_i, \mathsf{s}_j)$.*
2. *The induced k-path $\pi^\mathcal{I} = s_0 \dots s_k$ is a finite path in M.*
3. *If $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$ then the induced k-path $\pi^{l,\mathcal{I}} = s_0 \dots s_{l-1}(s_l \dots s_k)^\omega$ is an infinite path in M.*

The following two definitions give the translation of an LTL formula ϕ into a predicate encoding following an approach similar to structural clause form translations: a new predicate symbol is first introduced to represent each subformula, here denoted by $\Theta_\phi(X)$ in Definition 8, and then a set of constraints, given in Definition 9, are added to give $\Theta_\phi(X)$ its intended meaning.

Definition 8. We define the *symbolic representation* of an LTL formula γ, a predicate formula $\Theta_\gamma(X)$, as follows:

$$\Theta_\top(X) = \top \qquad \Theta_\bot(X) = \bot$$
$$\Theta_p(X) = \mathsf{p}(X) \qquad \Theta_{\neg p}(X) = \neg\mathsf{p}(X)$$
$$\Theta_{\psi \wedge \phi}(X) = \Theta_\psi(X) \wedge \Theta_\phi(X) \qquad \Theta_{\psi \vee \phi}(X) = \Theta_\psi(X) \vee \Theta_\phi(X)$$
$$\Theta_{\mathbf{X}\phi}(X) = \mathsf{next}_\phi(X) \qquad \Theta_{\mathbf{F}\phi}(X) = \mathsf{evently}_\phi(X)$$
$$\Theta_{\psi \mathbf{W} \phi}(X) = \mathsf{weak}_{\psi,\phi}(X)$$

where $\mathsf{next}_\phi(X)$, $\mathsf{evently}_\phi(X)$ and $\mathsf{weak}_{\psi,\phi}(X)$ are fresh new predicates, not already in $\mathcal{P}_\mathcal{V}$, introduced as needed for subformulae of γ. □

Definition 9. For every pair of LTL formulae ψ, ϕ and a value $k \geq 0$, we define the following sets of constraints:

$\Phi^k_{\mathbf{X}\phi}$: x1: $\mathsf{next}_\phi(X) \wedge \mathsf{trans}(X, Y) \rightarrow \Theta_\phi(Y)$
x2: $\mathsf{next}_\phi(\mathsf{s}_k) \rightarrow \mathsf{hasloop}$

$\Phi^k_{\mathbf{F}\phi}$: f1: $\mathsf{evently}_\phi(X) \rightarrow \mathsf{event}_\phi(X, \mathsf{s}_0) \vee \cdots \vee \mathsf{event}_\phi(X, \mathsf{s}_k)$
f2: $\mathsf{event}_\phi(X, Y) \rightarrow \Theta_\phi(Y)$
f3: $\mathsf{event}_\phi(X, Y) \wedge \mathsf{less}(Y, X) \rightarrow \mathsf{hasloop}$
f4: $\mathsf{event}_\phi(X, Y) \wedge \mathsf{less}(Y, X) \wedge \mathsf{trans}(\mathsf{s}_k, L) \wedge \mathsf{less}(Y, L) \rightarrow \bot$

$\Phi^k_{\psi\mathbf{W}\phi}$: w1: $\mathsf{weak}_{\psi,\phi}(X) \rightarrow \Theta_\phi(X) \vee \mathsf{xweak}_{\psi,\phi}(X)$
w2: $\mathsf{xweak}_{\psi,\phi}(X) \wedge \mathsf{trans}(X, Y) \rightarrow \mathsf{weak}_{\psi,\phi}(Y)$
w3: $\mathsf{xweak}_{\psi,\phi}(X) \rightarrow \Theta_\psi(X)$
w4: $\mathsf{xweak}_{\psi,\phi}(\mathsf{s}_k) \rightarrow \mathsf{hasloop}$

Again $\mathsf{event}_\phi(X, Y)$ and $\mathsf{xweak}_{\psi,\phi}(X)$ are fresh new predicates not in $\mathcal{P}_\mathcal{V}$.

We finally introduce the set of *structural definitions* of an LTL formula γ (with depth k), denoted by $|[\,\gamma, k\,]|$, as the union of the sets Φ^k_ϕ for every temporal subformula ϕ of the original γ. □

Later in Proposition 2 we show how the models of such formulae relate to the k-paths satisfying an LTL formula. We need first to introduce the concept of a *rolling function* which will be used as a tool in the proof of such proposition.

Definition 10. Given a k-path π we define its *rolling function* δ, a function defined for every $0 \leq i < |\pi|$ and with range $\{0, \ldots k\}$, as follows:

– If π is of the form $s_0 \ldots s_{l-1}(s_l \ldots s_k)^\omega$, then

$$\delta(i) = \begin{cases} i & i \leq k \\ l + [(i - l) \bmod (k + 1 - l)] & \text{otherwise .} \end{cases}$$

– Otherwise, if $\pi = s_0 \ldots s_k$, then $\delta(i) = i$ for every $0 \leq i < |\pi|$. □

changed!

The rolling function is a notational convenience used to unfold an infinite k-path $\pi = s_0 \ldots s_{l-1}(s_l \ldots s_k)^\omega$ as the sequence $\pi = s_{\delta(0)} s_{\delta(1)} \ldots$, without explicitly showing the loop. We emphasise the fact that the rolling function is defined only when $0 \leq i < |\pi|$; in particular, if π is finite, the function is not defined for indices outside of the path. Also notice that, for both finite and infinite paths, the rolling function acts as the identity for all i with $0 \leq i \leq k$. Moreover, for $0 \leq i < |\pi|$, it is always the case that $s_i = s_{\delta(i)}$; in fact, the following stronger result holds.

Lemma 2. *Let π be a k-path, ϕ an LTL formula, $i < |\pi|$ and δ the rolling function of π. Then it follows that $\pi \models_i \phi$ if and only if $\pi \models_{\delta(i)} \phi$.*

We can now prove one of the main propositions, which shows how from models of the encoded formula, one can obtain a k-path in the given Kripke structure that, moreover, satisfies the original LTL formula at a particular state.

Proposition 2. *Let M be a Kripke structure, γ an LTL formula, and $\mathcal{I}$ a model of the formula $|[\,M, k\,]| \cup |[\,\gamma, k\,]|$ with domain $D = \{\mathsf{s}_0, \ldots, \mathsf{s}_k\}$. We define a path π according to the following two cases:*

1. *If* $\mathcal{I} \models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$*, for some* $0 \leq l \leq k$*, then let* $\pi = \pi^{l,\mathcal{I}}$ *for any such* l*.*
2. *If* $\mathcal{I} \not\models \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_l)$*, for every* $0 \leq l \leq k$*, then let* $\pi = \pi^{\mathcal{I}}$*.*

Let $i < |\pi|$*, and let* δ *be the rolling function of* π*. If* $\mathcal{I} \models \Theta_\gamma(\mathsf{s}_{\delta(i)})$ *then* $\pi \models_i \gamma$*.*

The previous proposition shows that, under the given assumptions, if an interpretation $\mathcal{I} \models \Theta_\phi(\mathsf{s}_{\delta(i)})$ then there is a path π, determined by $\mathcal{I}$, such that $\pi \models_i \phi$. Note, however, that the converse is not always true, e.g. $\mathcal{I} \not\models \Theta_\phi(\mathsf{s}_{\delta(i)})$ does not necessarily imply $\pi \not\models_i \phi$ for the possible induced paths.

Additional constraints could be added to the set $|[\,\phi, k\,]|$ in order to make the converse hold but, since we are mostly interested in satisfiability of the LTL formulae, this is not required for the correctness of our main result. Whether the addition of such constraints would be helpful for the solvers to find solutions more quickly, is an interesting question for further research.

What we do need to show is that, if there is a path that satisfies an LTL formula, we can also find an interpretation that satisfies its symbolic representation. The following definition shows how to build such interpretation and later, in Proposition 3, we prove it serves the required purpose.

Definition 11. Let π be a k-path and δ its rolling function. We define an interpretation $\mathcal{I}^\pi$ with domain $D = \{\mathsf{s}_0, \ldots, \mathsf{s}_k\}$, for every $\mathsf{s}_i, \mathsf{s}_j \in D$ and pair of LTL formulae ψ, ϕ, as follows:

$\mathcal{I}^\pi \models \mathsf{p}(\mathsf{s}_i)$	iff	$p \in s_i$, for $p \in \mathcal{V}$.
$\mathcal{I}^\pi \models \mathsf{less}(\mathsf{s}_i, \mathsf{s}_j)$	iff	$i < j$.
$\mathcal{I}^\pi \models \mathsf{succ}(\mathsf{s}_i, \mathsf{s}_j)$	iff	$i + 1 = j$.
$\mathcal{I}^\pi \models \mathsf{trans}(\mathsf{s}_i, \mathsf{s}_j)$	iff	$\delta(i+1) = j$.
$\mathcal{I}^\pi \models \mathsf{hasloop}$	iff	π is an infinite path.
$\mathcal{I}^\pi \models \mathsf{next}_\phi(\mathsf{s}_i)$	iff	$\pi \models_i \mathbf{X}\phi$.
$\mathcal{I}^\pi \models \mathsf{evently}_\phi(\mathsf{s}_i)$	iff	$\pi \models_i \mathbf{F}\phi$.
$\mathcal{I}^\pi \models \mathsf{event}_\phi(\mathsf{s}_i, \mathsf{s}_j)$	iff	$\pi \models_j \phi$ and there is a $j' \geq i$ with $\delta(j') = j$.
$\mathcal{I}^\pi \models \mathsf{weak}_{\psi,\phi}(\mathsf{s}_i)$	iff	$\pi \models_i \psi \mathbf{W} \phi$.
$\mathcal{I}^\pi \models \mathsf{xweak}_{\psi,\phi}(\mathsf{s}_i)$	iff	$\pi \models_i \psi \mathbf{W} \phi \wedge \neg\phi$.

□

Proposition 3. *Let* π *be a* k*-path in a Kripke structure* M*, and* δ *its rolling function. Also let* γ *be an arbitrary LTL formula, and let* $i < |\pi|$*.*

1. $\mathcal{I}^\pi \models \Theta_\gamma(\mathsf{s}_{\delta(i)})$ *iff* $\pi \models_i \gamma$*,*
2. $\mathcal{I}^\pi \models |[\,M, k\,]| \cup |[\,\gamma, k\,]|$*.*

With this results being put in place we can now show, in Theorem 2, how the problem of testing the satisfiability of an LTL formula in a Kripke structure can be translated into the problem of checking satisfiability of predicate formulae.

Definition 12. Let M be a Kripke structure, ϕ an LTL formula and $k \geq 0$. The *predicate encoding* of M and ϕ (with depth k), denoted by $|[\,M, \phi, k\,]|$, is defined as the set of constraints $|[\,M, k\,]| \cup |[\,\phi, k\,]| \cup \{\Theta_\phi(\mathsf{s}_0)\}$. □

Theorem 2. *Let* ϕ *be an LTL formula, and* M *a Kripke structure.*

1. ϕ *is satisfiable in* M *iff* $|[\,M, \phi, k\,]|$ *is satisfiable for some* $k \geq 0$*.*
2. ϕ *is valid in* M *iff* $|[\,M, \mathsf{NNF}(\neg\phi), k\,]|$ *is unsatisfiable for every* $k \geq 0$*.*

3.1 Implicit bound encoding

As can be seen in Definition 9, the encoding just presented makes explicit use of the bound k in order to build the symbolic representation of an LTL formula. Notice that, in particular, a constraint of size $O(k)$ is created for every subformula of the form $\mathbf{F}\phi$ of the property to be checked. In this section we present an alternate encoding, which only uses the bound in an implicit way.

Definition 13. Given pair of LTL formulae ψ, ϕ, we define the following sets of constraints:

$\Phi'_{\mathbf{F}\phi}$: f1': $\mathsf{evently}_\phi(X) \rightarrow \Theta_\phi(X) \vee \mathsf{xevently}_\phi(X)$
f2': $\mathsf{xevently}_\phi(X) \wedge \mathsf{succ}(X,Y) \rightarrow \mathsf{evently}_\phi(Y)$
f3': $\mathsf{xevently}_\phi(X) \wedge \mathsf{last}(X) \rightarrow \mathsf{hasloop}$
f4': $\mathsf{xevently}_\phi(X) \wedge \mathsf{last}(X) \wedge \mathsf{trans}(X,Y) \rightarrow \mathsf{evently2}_\phi(Y)$
f5': $\mathsf{evently2}_\phi(X) \rightarrow \Theta_\phi(X) \vee \mathsf{xevently2}_\phi(X)$
f6': $\mathsf{xevently2}_\phi(X) \wedge \mathsf{succ}(X,Y) \rightarrow \mathsf{evently2}_\phi(Y)$
f7': $\mathsf{xevently2}_\phi(X) \wedge \mathsf{last}(X) \rightarrow \bot$

The sets $\Phi'_{\mathbf{X}\phi}$ and $\Phi'_{\psi\mathbf{W}\phi}$ are identical to $\Phi^k_{\mathbf{X}\phi}$ and $\Phi^k_{\psi\mathbf{W}\phi}$, except for the following constraints which replace x2 and w4 respectively.

$\Phi'_{\mathbf{X}\phi}$: x2': $\mathsf{next}_\phi(X) \wedge \mathsf{last}(X) \rightarrow \mathsf{hasloop}$

$\Phi'_{\psi\mathbf{W}\phi}$: w4': $\mathsf{xweak}_{\psi,\phi}(X) \wedge \mathsf{last}(X) \rightarrow \mathsf{hasloop}$

We finally introduce the set of *implicit structural definitions* of an LTL formula γ, denoted simply by $|[\,\gamma\,]|$, as the union of the sets Φ'_ϕ for every temporal subformula ϕ of the original γ. □

Note that the newly defined sets Φ'_ϕ, do not explicitly use the value of the bound k anymore. We replaced the explicit references to s_k with a predicate $\mathsf{last}(X)$ which should be made true for the constant symbol representing the last state. Moreover, since the size of $\Phi'_{\mathbf{F}\phi}$ is constant, the size of the encoding $|[\,\gamma\,]|$ is now linear with respect to the size of γ.

The k-paths that satisfy an LTL formula ϕ can therefore now be captured with the set of constraints $|[\,k\,]| \cup \{\mathsf{last}(\mathsf{s}_k)\} \cup |[\,\phi\,]| \cup \{\Theta_\phi(\mathsf{s}_0)\}$. This representation is convenient since it breaks the encoding in two independent parts, one depending on the bound only and the other on the LTL formula only. Moreover it has a size of $O(n+k)$ where n is the size of the original temporal formula.

A complete instance of the bounded model checking problem would be then represented, analogous to Definition 12, as

$$|[\,M,\phi,k\,]|^* = |[\,M\,]| \cup |[\,k\,]| \cup \{\mathsf{last}(\mathsf{s}_k)\} \cup |[\,\phi\,]| \cup \{\Theta_\phi(\mathsf{s}_0)\}$$

and, for such set of constraints, the statement of Theorem 2 also holds.

changed! This encoding is particularly useful when searching for counterexamples in an incremental setting, since both the system description and the temporal formula have to be encoded only once. Just the small set $|[\,k\,]| \cup \{\mathsf{last}(\mathsf{s}_k)\}$ needs to be updated while testing for increasing bounds. If using a model finder that supports incremental solving features, then one only needs to add $\mathsf{succ}(\mathsf{s}_k, \mathsf{s}_{k+1})$ and replace $\mathsf{last}(\mathsf{s}_k)$ with $\mathsf{last}(\mathsf{s}_{k+1})$.

3.2 Logarithmic encoding of states

As can be seen in the previous section, the only part of the translation where there is an increase of size with respect to the input is in $|[\,k\,]|$, because of the series of facts of the form $\mathsf{succ}(\mathsf{s}_i, \mathsf{s}_{i+1})$ and the constraint

$$\mathsf{hasloop} \rightarrow \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_0) \vee \cdots \vee \mathsf{trans}(\mathsf{s}_k, \mathsf{s}_k)\ . \tag{1}$$

This group of constraints, which is of size $O(k)$, can be more compactly encoded by representing the names of states in binary notation. For this we introduce a pair of constant symbols $\{\mathsf{b0}, \mathsf{b1}\}$ so that we can write, for example when $k = 2^4$, the following definition for the succ predicate:

$$\begin{array}{l}
\mathsf{succ}(\ X_3, X_2, X_1, \mathsf{b0}\ ,\ X_3, X_2, X_1, \mathsf{b1}\ ) \\
\mathsf{succ}(\ X_3, X_2, \mathsf{b0}\ , \mathsf{b1}\ ,\ X_3, X_2, \mathsf{b1}\ , \mathsf{b0}\ ) \\
\mathsf{succ}(\ X_3, \mathsf{b0}\ , \mathsf{b1}\ , \mathsf{b1}\ ,\ X_3, \mathsf{b1}\ , \mathsf{b0}\ , \mathsf{b0}\ ) \\
\mathsf{succ}(\ \mathsf{b0}\ , \mathsf{b1}\ , \mathsf{b1}\ , \mathsf{b1}\ ,\ \mathsf{b1}\ , \mathsf{b0}\ , \mathsf{b0}\ , \mathsf{b0}\ )
\end{array}$$

In general we only need $w = \lceil \log k \rceil$ constraints, with a total size of $O(\log^2 k)$. On the other hand, the constraint (1) can be rewritten as:

$$\begin{array}{l}
\mathsf{hasloop} \rightarrow \mathsf{loopafter}(\overline{\mathsf{b0}}) \\
\mathsf{loopafter}(\overline{X}) \wedge \mathsf{last}(\overline{Y}) \rightarrow \mathsf{trans}(\overline{Y}, \overline{X}) \vee \mathsf{xloopafter}(\overline{X}) \\
\mathsf{xloopafter}(\overline{X}) \wedge \mathsf{succ}(\overline{X}, \overline{Y}) \rightarrow \mathsf{loopafter}(\overline{Y}) \\
\mathsf{xloopafter}(\overline{X}) \wedge \mathsf{last}(\overline{X}) \rightarrow \bot
\end{array}$$

where $\overline{\mathsf{b0}}$ is a string of w symbols $\mathsf{b0}$, $\overline{X} = X_{w-1}, \ldots, X_0$ and similarly for $\overline{Y}$.

One also has to replace everywhere else occurrences of s_0 with $\overline{\mathsf{b0}}$, the constant symbol s_k with its binary representation (e.g. for $k = 13$ use $\mathsf{b1}, \mathsf{b1}, \mathsf{b0}, \mathsf{b1}$), and variables such as X and Y with the corresponding $\overline{X}$ or $\overline{Y}$. The resulting set of constraints, which we denote by $|[\,M, k, \phi\,]|^b$, is of size $O(n \log k + \log^2 k)$, where n is the compound size of M and ϕ, and satisfies the statement of Theorem 2.

4 Encoding of the system description

Generating an instance of the bounded model checking problem requires three parameters as input: a system description M, a temporal formula ϕ and a bound k. In the previous section we showed how to encode an LTL formula as a predicate formula (w.r.t. the bound), but we generally assumed that the system (a Kripke structure M) was already symbolically described.

In this section we deal with how a system, which is originally given in some industry standard format suitable to describe software/hardware components, can be also be encoded in the form of a predicate formula. An advantage of using a predicate rather than a propositional encoding is that important features for component development, such as the ability to describe systems in a modular and hierarchical way, can be directly represented in the target language. There

is no need, for example, to perform a flattening phase to create and instantiate all modules of a system description before doing the actual encoding.

We will show now, by means of an example, how a system described in the smv language can be succinctly and naturally encoded within the effectively propositional fragment. Although we would prefer to formally define the fragment of smv considered here, the number of different smv variants and the lack of documentation on the formal semantics in existing implementations made this task particularly difficult. Anyway, the explanation of the ideas presented in this section is always general enough so that they can be applied to other arbitrary systems, not only the one in the example, and even implemented to be performed in an automated way. changed!

For our running example we consider a distributed mutual exclusion (DME) circuit first described by Martin [8] and then made available in the `smv` format with the distribution of the NuSMV model checker [3]. The system description is fragmented in a number of modules, each being a separate unit specifying how a section of the system works. The DME, for example, organises modules in a hierarchical way: the most basic modules are *gates* which perform simple logical operations, then a number of gate modules are replicated and assembled together to form the module of a *cell*, finally a number of cells are also replicated and linked together in the *main* module which represents the entire system.

4.1 Module variables

A module usually defines a number of variables and describe how their values change in time. In the DME example, a typical gate module looks like:

```
module and-gate(in1,in2)
var
  out: boolean;
assign
  init(out) := 0;
  next(out) := (in1 & in2) union out;
```

This is a module named 'and−gate' which defines two boolean variables as input ('in1' and 'in2') and an output boolean variable ('out'). The initialisation part causes the output of all 'and−gate' instances to hold the value zero (i.e. false) when the system starts to execute. At each step the module nondeterministically chooses to compute the logical and of its inputs and update the output, or keep the output from the last clock cycle.[1] Note that this is the model of an asynchronous logic gate; fairness constraints (which can also be encoded as LTL formulae) could be added to ensure, for example, that the gate eventually computes the required value.

In the symbolic description we represent each of these variables with a predicate symbol such as, in this particular example, $\mathsf{and\_gate\_in1}(I_1, I_2, X)$. The

[1] The '**union**' operator in smv effectively creates a set out of its two operands and nondeterministically chooses an element of the set as the result of the expression.

variable name is prefixed with the module name so that variables of different modules do not interfere with each other. Since, moreover, several instances of the 'and−gate' can be created, the first arguments I_1, I_2 serve to distinguish among such instances, the following section explains this in more detail. The last argument X represents a time step within the execution trace. Using this naming convention, the module can then be described as follows:

$$
\begin{aligned}
&\neg \mathsf{and\_gate\_out}(I_1, I_2, \mathsf{s}_0) \\
&\mathsf{trans}(X, Y) \rightarrow \\
&\quad (\mathsf{and\_gate\_out}(I_1, I_2, Y) \leftrightarrow \mathsf{and\_gate\_in1}(I_1, I_2, X) \wedge \mathsf{and\_gate\_in2}(I_1, I_2, X)) \\
&\quad \vee (\mathsf{and\_gate\_out}(I_1, I_2, Y) \leftrightarrow \mathsf{and\_gate\_out}(I_1, I_2, X))
\end{aligned}
$$

Note that, although the original smv description distinguishes between inputs and outputs of the module, our proposed encoding does not need to.

4.2 Submodel instances

Modules can also create named instances of other modules and specify how its own variables and the variables of the its submodule instances relate to each other. There is also one designated 'main' module, an instance of which represents the entire system to verify. One has to distinguish between the notions of a module (the abstract description of a component) and its possibly many module instances, which actually conform the complete system. In our running example, the DME circuit, part of the definition of a cell module looks like:

```
module cell(left,right,token)
var
  ack: boolean;
  c: and−gate(a.out, !left.ack);
  d: and−gate(b.out, !u.ack)
  ⋮
```

Here two submodule instances 'c' and 'd' are created, both instances of the 'and−gate' module. The elements 'a, b: mutex_half' and 'u: user' are instances of other modules also created within the cell, with definitions of other internal variables such as 'out' and 'ack'. The elements 'left' and 'right' are references to other 'cell' instances, these are explained later in the following section.

Symbolically, we can describe the relations between the inputs and outputs of these modules using the constraints:

$$
\mathsf{cell\_left}(I, J) \rightarrow \begin{array}{l}
\mathsf{and\_gate\_in1}(I, \mathsf{c}, X) \leftrightarrow \mathsf{mutex\_half\_out}(I, \mathsf{a}, X) \\
\mathsf{and\_gate\_in2}(I, \mathsf{c}, X) \leftrightarrow \neg \mathsf{cell\_ack}(J, X) \\
\mathsf{and\_gate\_in1}(I, \mathsf{d}, X) \leftrightarrow \mathsf{mutex\_half\_out}(I, \mathsf{b}, X) \\
\mathsf{and\_gate\_in2}(I, \mathsf{d}, X) \leftrightarrow \neg \mathsf{user\_ack}(I, \mathsf{u}, X)
\end{array} \tag{2}
$$

Here the variable I stands for a particular cell instance, the second argument of the predicates is now filled in with the instance names of the different modules.

In general, if a module M_1 creates instances of a module M_2, we say that M_2 is a *submodule* of M_1. The submodule relation must then create a directed acyclic graph among the modules of a system; and the *submodule depth* of a module is the length of the longest path that can reach it from the designated 'main' module. The depth of the 'main' module, for example, is always 0; and the depth of a module is strictly less than the depth of its submodules.

In a module of depth d we will therefore use $d+1$ arguments in the predicates that represent the module's boolean variables. The last argument always denotes time, and the interpretation of the other d arguments is the string of names that represent each created instance in a chain of submodules. Consider for example the 'out' variable of a module 'some−gate' which corresponds to an instance with the fully qualified name of 'main.sub1.sub2.sub3.sub4.out'; symbolically we would represent such variable with the predicate

$$\mathsf{some\_gate\_out}(\mathsf{sub1}, \mathsf{sub2}, \mathsf{sub3}, \mathsf{sub4}, X)\,.$$

Finally note that instances of the same module could be reached from the main module by paths of different lengths.[2]. Consider for example a module of depth d that creates an instance named 'sub' of another module of depth d'; if a sequence of constant symbols $\mathsf{m}_1, \ldots, \mathsf{m}_d$ is used to identify an instance of the first module, then the sequence of d' constant symbols $\mathsf{m}_1, \ldots, \mathsf{m}_d, \ldots, \mathsf{o}, \ldots, \mathsf{sub}$ —where a number of dummy constant symbols 'o' (unused anywhere else) serve as padding to get the required length— is used to identify the second.

4.3 Module references

Another feature of the smv language is that modules can receive references to other modules as parameters (e.g. ' left ' and 'right' in the cell example). This feature is encoded introducing a new predicate, c.f. $\mathsf{cell\_left}(I, J)$ in (2), that establishes these relation between the two modules. References are used in our running example to communicate three different cells 'e−1', 'e−2' and 'e−3':

```
module main
var
  e-3: process cell(e-1,e-2,1);
  ⋮
```

which is encoded as: $\{\mathsf{cell\_left}(\mathsf{e\_3}, \mathsf{e\_1}), \mathsf{cell\_right}(\mathsf{e\_3}, \mathsf{e\_2}), \mathsf{cell\_token}(\mathsf{e\_3}, X)\}$. In general, the reference from a 'module1' to another 'module2' is encoded as:

$$\mathsf{module1\_link}(\overline{I}, \overline{J}) \rightarrow \mathsf{module1\_var1}(\overline{I}, X) \leftrightarrow \mathsf{module2\_var2}(\overline{J}, X)$$

where $\overline{I}$ and $\overline{J}$ are sequences of variables of appropriate lengths according to the depths of each module, and 'link' is the local name which the first module uses to reference the second. Compare this with the relevant constraint in (2).

[2] Consider a module 'm1' that creates instances of 'm2' and 'm3', but 'm2' also creates instances of 'm3'. As long as the *submodule* relation is acyclic, this is possible.

4.4 Enumerated types

Finally, another common feature of component description languages is the use of enumerated types, e.g. 'colour: {red, green, blue}'. Using standard encodings, such variables are represented with an additional argument to denote the value currently hold. Also, a number of constraints have to be added in order to ensure that one (and only one) value of an enumerated variable holds at a time.

5 Conclusions and future work

In this paper we presented different strategies to encode instances of the bounded model checking problem as a predicate formula in the Bernays-Schönfinkel class. We showed a translation which, given a linear temporal logic formula and a bound k, produces a set of constraints whose models represent all the possible paths (of bounded length k) which satisfy the given property. We also discussed how to further improve this translation and generate an output of size $O(n+k)$ where n is the size of the input LTL formula. The translation is also further improved by using a binary representation to denote the states.

We then proceeded to show how to efficiently describe transition systems as effectively propositional formulae, and demonstrated how many features commonly found in software/hardware description languages are succinctly and naturally encoded within our target language. Most significantly, modular and hierarchical system descriptions are directly encoded without a significant increase in the size; unlike propositional encodings where a preliminary, and potentially exponential, flattening phase needs to be applied to the system description.

We are also currently working in the development of a tool that —taking as input a `smv` description, an LTL formula, and a bound k— produces an EPR formula in the `tptp` format suitable for use with effectively propositional and first-order reasoners.[3] Directions for future work include the extension to more general forms of temporal logics (such as μTL), the inclusion of more features to describe systems (such as arrays and arithmetic) and the application of similar encoding techniques to other suitable application domains.

References

[1] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS '99*, volume 1579 of *LNCS*, 1999.

[2] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5:5), 2006.

[3] A. Cimatti, E. Clarke, F. Giunchiglia, M. Pistore, Marco Roveri, R. Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic

[3] The developed tool and a number of generated benchmarks are publicly available at `http://www.cs.man.ac.uk/~navarroj/eprbmc`.

model checking. In *CAV'02*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[4] 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 *CAV'01*, volume 2102 of *LNCS*, pages 436–453. Springer, 2001.

[5] Martin Davis, George Logemann, and Dondald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[6] Markus Jehle, Jan Johannsen, Martin Lange, and Nicolas Rachinsky. Bounded model checking for all regular properties. *Electr. Notes Theor. Comput. Sci.*, 144(1):3–18, 2006.

[7] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple bounded LTL model checking. In *FMCAD'04*, volume 3312 of *LNCS*, pages 186–200. Springer, 2004.

[8] Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration*, 1985.

[9] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[10] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7:156–173, April 2005.

[11] O. Strichman. Accelerating bounded model checking for safety properties. *Formal Methods in System Design*, 24(1):5–24, 2004.

[12] G. Sutcliffe and C. Suttner. The state of casc. *AI Communications*, 19(1): 35–48, 2006.

[13] G. Sutcliffe and C.B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.