First-Order-CTL Model Checking*

Jürgen Bohn[1] Werner Damm[1] Orna Grumberg[2] Hardi Hungar[1] Karen Laster[2]

[1] {bohn,damm,hungar}@OFFIS.Uni-Oldenburg.DE
[2] {orna,laster}@CS.Technion.AC.IL

Abstract. This work presents a first-order model checking procedure that verifies systems with large or even infinite data spaces with respect to first-order CTL specifications. The procedure relies on a partition of the system variables into *control* and *data*. While control values are expanded into BDD-representations, data values enter in form of their properties relevant to the verification task. The algorithm is completely automatic. If the algorithm terminates, it has generated a first-order verification condition on the data space which characterizes the system's correctness. Termination can be guaranteed for a class that properly includes the data-independent systems, defined in [10].
This work improves [5], where we extended *explicit* model checking algorithms. Here, both the control part and the first-order conditions are represented by BDDs, providing the full power of symbolic model checking for control aspects of the design.

1 Introduction

Symbolic model checking is currently one of the most successful formal methods for hardware verification. It can be applied to verify or debug designs from industrial practice, in particular when it is combined with techniques for abstraction and compositional reasoning. However, the majority of designs will escape even those combined approaches if detailed reasoning about the data path is required. This is because the data part is often too large (or even infinite) and too complicated.

To reduce the data-induced complexity, we combine symbolic model checking for the control part of a design with a generation of first-order verification conditions on data in an algorithm we call *first-order model checking*. Thereby, we achieve a separation of the verification task, where the verification conditions on the data which are computed by our algorithm may be handled afterwards by any appropriate means, for instance by a theorem prover. The algorithm behaves like a symbolic model checker if all system variables are treated as control variables, offering the verification condition generation as an option if the data complexity turns out to be too high.

The algorithm is intended to be applied to a class of applications which permit a clear separation between *control* and *data*. Examples of such systems are embedded control applications, where the control part governs the interaction between the controller and the controlled system, depending in its actions only on some flags derived from the data, and generating data output by finitely many computations. Others are processors with nontrivial data paths.

For such applications, the algorithm will terminate, while in general it will not. The class of systems for which it works properly includes Wolper's data-independent systems [10], where there is neither testing nor computing on data values. Either testing or computing can be permitted, but unrestricted use of both may lead to nontermination

* This research has been funded in part by the MWK under No. 210.3 - 70631/9 - 99 - 14/96.

of our procedure. Due to space limitations, an extensive discussion of the termination issue is not given in this paper.

On a technical level, we take FO-CTL, a first-order version of CTL, as our specification logic. Programs appear in an abstract representation as *first-order Kripke structures*. These extend "ordinary" Kripke structures by transitions with conditional assignments, capturing the effect of taking a transition on an underlying possibly infinite state space induced from a set of typed variables.

Our algorithm incrementally computes for each state of the first-order Kripke structure and each control valuation a first-order predicate over the data variables which characterizes validity of the FO-CTL formula to be proven. The algorithm is developed from a standard CTL symbolic model-checking procedure, not the automata-theoretic approach from [2], as that one does not permit an extension which generates the annotations along the way.

The work presented here extends the previous paper [5] in two respects. First, we now represent and generate first-order annotations symbolically as BDDs. Due to this representation, we can avoid intermediate calls to a theorem prover, as we get, for instance, propositional simplifications "for free". Also, the control valuations are not expanded explicitly, enabling us to cope with much larger control parts. Second, the procedure can now cope with unboundedly many inputs. The addition necessary for that will be pointed out during the description of the algorithm in Sec. 3.

The paper contains a description of our core algorithm, and demonstrates its operation for a simple example problem. There are extensions and optimizations of which we present only one, *control pruning*, which is one of the most important amendments.

Related work: Approaches based on abstraction like the ones in [4, 7] and, to some extent, in [6] try to reduce the state space to a small, resp., finite one, where the proof engineer is required to find suitable abstractions for program variables. In our approach, the verifier's main involvement is in deciding which variables to consider as control.

Using uninterpreted functions in order to model non-finite aspects of the system is more closely related to our work. In such approaches, uninterpreted functions represent terms or quantifier-free first-order formulas, whereas in our work a notion similar to uninterpreted functions represents first-order formulas with quantification.

In [8], systems with infinite memory are verified. States are represented as a combination of an explicit part for data variables and a symbolic part for control variables. Their algorithm, if it terminates, computes the set of reachable states. Thus, they can check only safety properties.

In [9], designs are described by quantifier-free formulas with equality, over uninterpreted functions. A BDD representation of the formulas enables symbolic satisfiability checking. This approach is used for checking design equivalence. No temporal specification logic is considered. There are similar approaches to check the uninterpreted equivalence step functions, e.g. [3].

In [11], data operations on abstract data variables are represented by uninterpreted functions. They check a restricted subset of a first-order linear-time temporal logic with fairness. Their logic allows limited nesting of temporal operators, thus, it is incomparable to our FO-CTL. FO-CTL, on the other hand, allows any nesting of CTL operators, and includes both existential and universal path quantifiers.

In [1], symbolic model checking with uninterpreted functions is used to verify an algorithm for out-of-order execution. They use a reference file (which is similar to our proposition table) to represent system behaviors. Their method is aimed at proving partial and total correctness of out-of-order executions while we verify general FO-CTL properties.

2 Semantical Foundation

2.1 First-Order Kripke Structures

The basic semantical domain is a Kripke structure, although we will assume that the programming language has its semantics in the more abstract domain of *first-order* Kripke structures.

A *Kripke structure* is a quintuple $(S, R, \mathcal{A}, L, I)$ where S is a not necessarily finite set of *states*, $R \subseteq S \times S$ is the *transition relation*, $\mathcal{A}$ is a set of *atoms*, $L : S \to \mathcal{P}(\mathcal{A})$ is the *labeling function* and $I \subseteq S$ is the set of *initial states*.

We write $s \to s'$ for $(s, s') \in R$. K_s is the structure with the set of initial states set to $\{s\}$. The set $Tr(K)$ of *paths* of a Kripke structure is the set of maximal sequences $\pi = (s_0, s_1, \ldots)$ with $s_0 \in I$ and $s_i \to s_{i+1}$. We use π_i to denote s_i. $|\pi|$ will denote the number of states in the path π, if π is finite. It will be ∞ otherwise.

We now prepare the definition of first-order Kripke structures. A data structure on which a program operates is given by a *signature* $\mathcal{S}$, which introduces typed symbols for constants and functions, and by an interpretation $\mathcal{I}$ which assigns a meaning to symbols, based on an interpretation $\mathcal{I}(\tau)$ of types τ by domains. We assume that the set of types include type `bool` and that there is an equality symbol "$=_\tau$" for each type τ.

For a signature $\mathcal{S}$ and a typed set of variables `V`, we denote by `T(V)` the set of terms and by `B(V)` the set of boolean expressions over `V`. If we enrich the set `T(V)` with a specific constant $?_\tau$ for each type τ, we get the set of *random* terms $\mathtt{T}^+\mathtt{(V)}$. For each set `V` of typed variables a set of type-respecting valuations $\mathcal{V}(\mathtt{V})$ is defined. If $\sigma \in \mathcal{V}(\mathtt{V})$ and an interpretation $\mathcal{I}$ are given, we get interpretations $\sigma(\mathtt{t})$ of terms.

To model inputs we consider *random* (or nondeterministic) assignments of the form `v:=?`, and call `v:=t` with `t`$\neq$`?` *regular* assignments. Let $\mathcal{I}(\mathtt{v:=t})$ be the semantics of an assignment as a binary relation on $\mathcal{V}(\mathtt{V})$, i.e.,

$$(\sigma, \sigma') \in \mathcal{I}(\mathtt{v:=t}) \;\Leftrightarrow_{\text{df}}\; \begin{cases} \sigma'(\mathtt{v}) = \sigma(\mathtt{t}),\ \mathtt{t} \in \mathtt{T(V)} \\ \sigma'(\mathtt{v}) \in \mathcal{I}(\tau),\ \mathtt{t} = ?_\tau \\ \sigma'(\mathtt{w}) = \sigma(\mathtt{w}),\ \mathtt{w} \neq \mathtt{v} \end{cases}$$

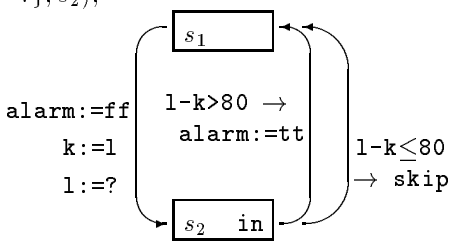
A set of assignments is *well-formed*, if the variables on the left-hand sides are pairwise distinct. The semantics of a well-formed set of assignments is the parallel execution of the single assignments. Let `WFA(V)` denote the set of well-formed assignment sets over a set of variables `V`. For a pair `(b,a)` of a condition `b` $\in$ `B(V)` and a well-formed assignment set `a` $\in$ `WFA(V)`, we set

$$\mathcal{I}(\mathtt{b,a}) =_{\text{df}} (\{\, \sigma \mid \sigma(\mathtt{b}) = \mathit{tt} \,\} \times \mathcal{V}(\mathtt{V})) \cap \mathcal{I}(\mathtt{a}).$$

A *first-order Kripke structure* over a signature $\mathcal{S}$ is a tuple $K = (S, \mathtt{V}, R, \mathcal{A}, L, I)$ where S is a finite set of *states*, `V` is a set of typed variables, $R \subseteq S \times \mathtt{B(V)} \times \mathtt{WFA(V)} \times S$ is the *transition relation*, $\mathcal{A}$ is a set of *atoms*, $L : S \to \mathcal{P}(\mathcal{A})$ is the *labeling function*, and $I \subseteq S \times \mathtt{B(V)}$ is the set of *initial states*.

In this paper we use first-order Kripke structures as our programming language. Intuitively, S is used to describe the (rather small set of) program locations, while the valuations $\mathcal{V}(\mathtt{V})$ might become rather big, or even infinite. For example, the following first-order structure K describes a controller that sets an alarm if a frequently read input value grows too fast.

$$
\begin{aligned}
S &=_{\text{df}} \{s_1, s_2\}\\
\mathtt{V} &=_{\text{df}} \{\mathtt{alarm:bool}, \mathtt{k,l:int}\}\\
R &=_{\text{df}} \{(s_1, \mathtt{tt}, \{\mathtt{alarm:=ff,\ k:=l,\ l:=?}\}, s_2),\\
&\qquad (s_2, \mathtt{l\text{-}k{>}80}, \{\mathtt{alarm:=tt}\}, s_1),\\
&\qquad (s_2, \neg\mathtt{l\text{-}k{>}80}, \{\}, s_1)\}\\
\mathcal{A} &=_{\text{df}} \{\mathtt{in}\}\\
L(s) &=_{\text{df}} \begin{cases} \{\mathtt{in}\} & \text{if } s = s_1\\ \{\ \} & \text{if } s = s_2 \end{cases}\\
I &=_{\text{df}} \{(s_1, \mathtt{alarm{=}ff})\}
\end{aligned}
$$



A first-order structure is called *finite* if all its components are finite sets. Finiteness of a first-order structure does not imply that it necessarily represents a finite Kripke structure. Depending on the interpretation, data domains may be infinite, in which case we have a finite description of an infinite-state Kripke structure. We define the semantics of a first-order Kripke structure by means of a regular Kripke structure.

Given an interpretation $\mathcal{I}$ of $\mathcal{S}$, a first-order structure K over $\mathcal{S}$ induces the Kripke structure $K' = \mathcal{I}(K)$ with the components

$$
\begin{aligned}
S' &=_{\text{df}} S \times \mathcal{V}(\mathtt{V})\\
(s_1, \sigma_1) \to (s_2, \sigma_2) &\Leftrightarrow_{\text{df}} s_1 \xrightarrow{\mathtt{b,a}} s_2 \wedge (\sigma_1, \sigma_2) \in \mathcal{I}(\mathtt{b,a})\\
\mathcal{A}' &=_{\text{df}} \mathcal{A} \cup \{\ \mathtt{v{=}d} \mid \mathtt{v} \in \mathtt{V}, \mathtt{d} \in \mathcal{I}(\tau(\mathtt{v}))\ \}\\
L'(s, \sigma) &=_{\text{df}} L(s) \cup \{\ \mathtt{v} = \sigma(\mathtt{v}) \mid \mathtt{v} \in \mathtt{V}\ \}\\
I' &=_{\text{df}} \mathcal{I}(I) {=}_{\text{df}} \{\ (s, \sigma) \in S \times \mathcal{V}(\mathtt{V}) \mid (s, \mathtt{b}) \in I \wedge \sigma(\mathtt{b}) = \mathit{tt}\ \}
\end{aligned}
$$

By $K \uparrow \mathtt{x}$ we denote the first-order structure resulting from K if the variable $\mathtt{x}$ is added to the variable set, after renaming a possibly already present $\mathtt{x} \in \mathtt{V}$. Since the added $\mathtt{x}$ does not appear in conditions or assignments, its value is arbitrary in initial states and never gets changed.

2.2 First-Order Temporal Logic

To specify properties of first-order structures, we use first-order computation tree logic, FO-CTL. It has the following negation-free syntax.

$$
\begin{aligned}
\phi &::= A \mid \overline{A} \mid \mathtt{b} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathbf{q}\,\mathtt{x}.\,\phi \mid \mathbf{Q}\,\psi\\
\psi &::= \mathbf{X}\phi \mid \phi\ \mathbf{U}\,\phi \mid \phi\ \mathbf{W}\,\phi
\end{aligned}
$$

where $\mathtt{A} \in \mathcal{A}$, $\mathtt{b} \in \mathtt{B(V)}$, $\mathtt{x} \in \mathtt{V}$, $\mathbf{q} \in \{\forall, \exists\}$ and $\mathbf{Q} \in \{\mathbf{A}, \mathbf{E}\}$.

First-order quantifiers are interpreted *rigidly*, i.e. the value is fixed for the evaluation of the formula in its scope, it does not change over time.

The semantics of a formula ϕ depends on an interpretation $\mathcal{I}$ of $\mathcal{S}$ and a first-order Kripke structure whose variables include all free variables of ϕ. The formula is interpreted over a pair $(s, \sigma) \in \mathcal{I}(K)$. For example, we define

$$
\begin{aligned}
(s,\sigma) &\models_{\mathcal{I}(K)} A/\overline{A} \Leftrightarrow_{\text{df}} A \in L(s,\sigma)/A \notin L(s,\sigma)\\
(s,\sigma) &\models_{\mathcal{I}(K)} \mathtt{b} \Leftrightarrow_{\text{df}} \sigma(\mathtt{b}) = tt\\
(s,\sigma) &\models_{\mathcal{I}(K)} \mathbf{q}\mathtt{x}.\phi \Leftrightarrow_{\text{df}} \mathbf{q}d \in \mathcal{I}(\tau(\mathtt{x}))\,.\,(s,\sigma\{d/\mathtt{x}\}) \models_{\mathcal{I}(K\uparrow\mathtt{x})} \phi \quad \mathbf{q} \in \{\forall,\exists\}\\
(s,\sigma) &\models_{\mathcal{I}(K)} \mathbf{A}\psi \Leftrightarrow_{\text{df}} \forall\pi \in \mathit{Tr}(\mathcal{I}(K)_{(s,\sigma)})\,.\,\pi \models_{\mathcal{I}(K)} \psi\\
\pi &\models_{\mathcal{I}(K)} \mathbf{X}\phi \Leftrightarrow_{\text{df}} |\pi| > 1 \wedge \pi_1 \models_{\mathcal{I}(K)} \phi\\
\pi &\models_{\mathcal{I}(K)} \phi_1\,\mathbf{W}\,\phi_2 \Leftrightarrow_{\text{df}} \text{for all } i < |\pi|, \pi_i \models_{\mathcal{I}(K)} \phi_1, \text{ or there is } j < |\pi|,\\
&\qquad\qquad \pi_j \models_{\mathcal{I}(K)} \phi_2, \text{ and for all } i < j,\ \pi_i \models_{\mathcal{I}(K)} \phi_1
\end{aligned}
$$

If $\sigma \in \mathcal{V}(\mathtt{V})$, we denote by $\sigma\{d/\mathtt{x}\}$ the valuation of $\mathtt{V} \cup \{\mathtt{x}\}$ which maps $\mathtt{x}$ to d and agrees further with σ on the valuation of $\mathtt{V}\backslash\{\mathtt{x}\}$. Formally, we lift the standard definition of correctness of a Kripke structure wrt. a CTL formula to the first-order case :

> Given an interpretation $\mathcal{I}$ of $\mathcal{S}$, a first-order structure K *satisfies* a FO-CTL formula ϕ, denoted $K \models \phi$, if for every $(s,\sigma) \in \mathcal{I}(I)$, $(s,\sigma) \models_{\mathcal{I}(K)} \phi$.

As in the following example ϕ, we use standard abbreviations in specifications. Expanding logical implication requires moving negation through a formula down to atoms. $\mathbf{AG}\,\phi_1$ is defined by $\mathbf{A}[\phi_1\,\mathbf{W}\mathit{ff}]$ as in CTL.

$$\phi =_{\text{df}} \forall\mathtt{x}\,\mathbf{AG}\,(\mathtt{in} \wedge \mathtt{l} = \mathtt{x} \rightarrow \mathbf{AXAX}\,(\mathtt{in} \wedge \mathtt{l\text{-}x} > 100 \rightarrow \mathbf{AX}\,(\mathtt{alarm{=}tt})))$$

Note that we use the first-order variable $\mathtt{x}$ to "store" the value of $\mathtt{l}$ through two next-steps such that a comparison between new and old input values becomes possible. In Section 4.3 we check the previously introduced first-order structure against this ϕ.

3 First-Order Model Checking

Model checking of FO-CTL formulas can be described as an iterative computation on the cross product of the first-order Kripke structure with a *tableau* of the formula. After defining this product structure, we will first describe a semantical version of this computation. Afterwards, we will define a syntactic procedure, and finally show how it may be realized by a symbolic, i.e., BDD-based, model checker.

3.1 The Product Structure

The model checking algorithm operates on a product of a first-order Kripke structure with a *tableau* Tab_ϕ of a FO-CTL formula ϕ . Tab_ϕ captures the dependencies between the validity of subformulas of ϕ as known from CTL. In the first-order case here, the subformulas are accompanied by the set of variables which are quantified within the context in ϕ .

Formally, let Tab_ϕ be the smallest set of nodes N which contains $(\phi, \emptyset)$ and is closed with respect to the relations $\rightarrow = \{\,((\mathbf{QX}\,\phi_1, \mathtt{W}), (\phi_1, \mathtt{W}))\,\}$ and $\mapsto$ defined by

- If $n = (\phi_1 \wedge / \vee\, \phi_2, \mathtt{W})$, then $n \mapsto (\phi_1, \mathtt{W})$ and $n \mapsto (\phi_2, \mathtt{W})$.
- If $n = (\mathbf{Q}[\phi_1\,\mathbf{U}/\,\mathbf{W}\,\phi_2], \mathtt{W})$, then $n \mapsto (\phi_2, \mathtt{W})$, $n \mapsto (\phi_1, \mathtt{W})$ and $n \mapsto \mathbf{QXQ}[\phi_1\,\mathbf{U}/\,\mathbf{W}\,\phi_2]), \mathtt{W})$.
- If $n = (\mathbf{q}\mathtt{x}.\,\phi_1, \mathtt{W})$, then $n \mapsto (\phi_1, \mathtt{W} \cup \{\mathtt{x}\})$.

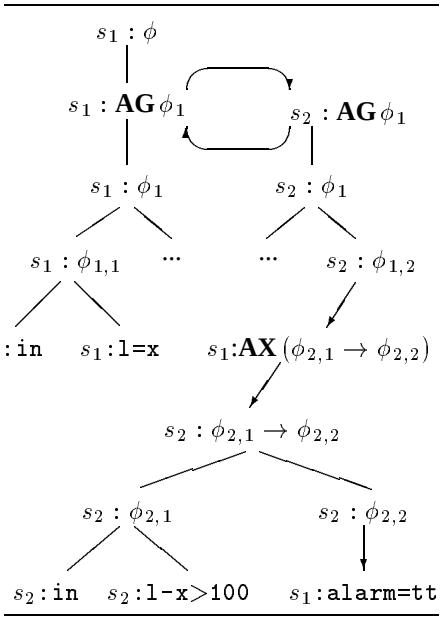
Given a finite first-order structure $K = (S, \mathtt{V}, R, \mathcal{A}, L, I)$ and a FO-CTL formula ϕ with tableau $\mathit{Tab}_\phi = (N, \mapsto, \rightarrow)$, we build the product structure $K \times \mathit{Tab}_\phi = (M, \mapsto', \rightarrow')$ as follows. We are assuming w.l.o.g. that no variable from $\mathtt{V}$ gets quantified within ϕ.

$$M =_{\text{df}} S \times N$$
$$(s,n) \mapsto' (s,n') \Leftrightarrow_{\text{df}} n \mapsto n'$$
$$(s,n) \xrightarrow{\mathtt{b,a}}{}' (s',n') \Leftrightarrow_{\text{df}} n \to n' \text{ and } s \xrightarrow{\mathtt{b,a}} s'$$

Let ϕ be the example specification from Sec. 2.2, $\phi = \forall \mathtt{x}\, \mathbf{AG}\,(\phi_1)$, with subspecifications

$$\begin{aligned}
\phi_1 &=_{\text{df}} \phi_{1,1} \to \phi_{1,2}\\
\phi_{1,1} &=_{\text{df}} \mathtt{in} \wedge \mathtt{l{=}x}\\
\phi_{1,2} &=_{\text{df}} \mathbf{AX}\,\mathbf{AX}\,(\phi_{2,1} \to \phi_{2,2})\\
\phi_{2,1} &=_{\text{df}} \mathtt{in} \wedge \mathtt{l\text{-}x} > \mathtt{100}\\
\phi_{2,2} &=_{\text{df}} \mathbf{AX}\,(\mathtt{alarm{=}tt})
\end{aligned}$$

A sketch of the product structure in our example is given on the right. The relation $\mapsto$ is drawn as a line, the conditional assignments labeling the arrows ($\to$) are missing, as well as the sets of quantified variables. Also, intermediate nodes have been removed from the cycle induced by $\mathbf{AG}\,(\phi_1) = \mathbf{A}[\phi_1\,\mathbf{W}\,\mathit{ff}]$.



Our model-checking procedures will operate on such cross products. The reader will note the similarity between cross products and the information flow in standard CTL model checking, which is opposite to the direction of the arrows.

The nodes labeled by a **W** (or **G**, as in the example) formula are called *maximal* nodes, those labeled by a **U** formula are *minimal* nodes. Nodes without successors are called *terminal* nodes. We will say that a node m' in the product structure is *below* some m, if m' is reachable from m via the union of the relations $\mapsto'$ and $\xrightarrow{\mathtt{b,a}}{}'$.

3.2 First-Order Model Checking, Semantically

Let the cross product between a first-order Kripke structure and the tableau of an FO-CTL formula ϕ be given. To each pair $(s, (\phi_1, \mathtt{W}))$ in the product structure the set of valuations σ of $\mathtt{V} \cup \mathtt{W}$ can be assigned s.t. $(s,\sigma) \models_{\mathcal{I}(K)} \phi_1$. Below, we describe a model-checking like procedure which, if it terminates, annotates each node with the correct set of variable valuations. The procedure works iteratively, starting with an initial annotation, and computing the new annotation of a node (s,n) from the annotations of its successor nodes (s',n').

The process starts by initializing all pairs with minimal nodes to $\emptyset$ and all those with maximal nodes $(\phi_1, \mathtt{W})$ to $\mathcal{V}(\mathtt{V} \cup \mathtt{W})$. Pairs with terminal nodes are initialized with the set of valuations satisfying the boolean formula after taking the state labeling into account, i.e.: if $n = (\phi_1, \mathtt{W})$ then

$$\mathit{ann}(s,n) =_{\text{df}} \{\, \sigma \in \mathcal{V}(\mathtt{V} \cup \mathtt{W}) \mid \sigma(\phi_1') = \mathit{tt} \,\},$$

where ϕ_1' results from ϕ_1 by replacing atomic formulas A by *tt* iff $A \in L(s)$ and by *ff* otherwise. In our example this means that $\mathit{ann}(s_1, (\mathtt{in,x})) = \emptyset$, $\mathit{ann}(s_2, (\mathtt{in,x})) = \mathcal{V}(\mathtt{V} \cup \{\mathtt{x}\})$, $\mathit{ann}(s_i, (\mathtt{term,x})) = \{\, \sigma \mid \sigma(\mathtt{term}) = \mathit{tt} \,\}$ for $i = 1, 2$ and $\mathtt{term} =$

`alarm=tt`, `l=x`, `l-x>100`. Nodes which are neither terminal nor maximal or minimal are not initialized.

After each step of the procedure, nodes for which the final annotation has been computed are identified. Initially, this set of nodes consists of the terminal nodes. The criterion for a node to have got its final annotation is that the annotations of all the nodes below it have been stable in the previous step.

It is an invariant of the process of annotation computation, that each annotation computed intermediately, satisfies that for maximal nodes it stays above the semantically correct annotation, and for minimal nodes, it stays below.

A step of the computation consists of updating the annotation of each node which has not yet got its final annotation. No update is done if some direct successor of a node is not yet annotated or if it is of a different mode (maximal/minimal) and not yet finally annotated. Otherwise, the update $\mathit{ann}'(s,n)$ is computed from the current annotations $\mathit{ann}(s',n')$ of the successors as follows ($n = (\mathbf{EX}\,\phi_1, \mathtt{W})$ omitted).

- If $n = (\phi_1 \wedge / \vee\, \phi_2, \mathtt{W})$, then $\mathit{ann}'(s,n) =_{\text{df}} \mathit{ann}(s,(\phi_1,\mathtt{W})) \cap / \cup\, \mathit{ann}(s,(\phi_2,\mathtt{W}))$.
- If $n = (\mathbf{AX}\,\phi_1, \mathtt{W})$, then
  $\mathit{ann}'(s,n) =_{\text{df}} \bigcap_{(s,n) \xrightarrow{\mathtt{b},\mathtt{a}} (s',n')} \{\, \sigma \mid \forall \sigma'.(\sigma,\sigma') \in \mathcal{I}(\mathtt{b},\mathtt{a}) \Rightarrow \sigma' \in \mathit{ann}(s',n') \,\}$.
- If $n = (\mathbf{Q}[\phi_1\,\mathbf{U}/\,\mathbf{W}\,\phi_2], \mathtt{W})$, and (s,n') is the successor node of (s,n), then
  $$\mathit{ann}'(s,n) =_{\text{df}} \mathit{ann}(s,n')\,.$$
- If $n = (\forall/\exists \mathtt{x}.\,\phi_1, \mathtt{W})$, and (s,n') is the successor node of (s,n), then
  $$\mathit{ann}'(s,n) =_{\text{df}} \{\, \sigma \in \mathcal{V}(\mathtt{V} \cup \mathtt{W}) \mid \forall/\exists d.\, \sigma\{\mathtt{x}/d\} \in \mathit{ann}(s,n') \,\}\,.$$

Proposition 1 states that this (semantical) annotation process is semantically sound.

Proposition 1. *If the annotation computation for $K \times \mathit{Tab}_\phi$ terminates, each node is annotated with the correct valuations, i.e., $\sigma \in \mathit{ann}(s,(\phi_1,\mathtt{W}))$ iff $(s,\sigma) \models_{\mathcal{I}(K\uparrow \mathtt{W})} \phi_1$.*

Proof (Sketch). Correctness for nodes which do not lie on cycles, i.e., ones which are neither minimal nor maximal, follows by induction (on the formula size, for instance). Using the invariant that minimal nodes are approximated from below and maximal nodes from above, we see that once a fixed point is reached for one such node, it is the minimal, resp., max., fixed point as required by the characterization of the operator.

Note, however, that while the procedure is sound, it need not terminate if the data domain is infinite. If the data domains are finite, though, it will terminate for the well-known monotonicity reasons.

As a consequence of Proposition 1, we see that a terminating annotation process yields a characterization of the correctness of the system.

Proposition 2. *If the semantical annotation process terminates, then $K \models \phi$ if and only if for every state s, if $(s,\sigma) \in \mathcal{I}(I)$ then $\sigma \in \mathit{ann}(s,(\phi,\emptyset))$ (meaning, by Proposition 1 that $(s,\sigma) \models_{\mathcal{I}(K)} \phi$).*

3.3 First-Order Model Checking, Syntactically

To turn the semantic procedure described above into a syntactic one, we replace the valuation sets $\mathit{ann}(s,n)$ by first-order formulas $\mathit{fo\text{-}ann}(s,n)$ describing them. We have to check that indeed each step of the computation has a syntactic counterpart.

The initialization of terminal nodes with the formula in the node directly gives us a first-order formula. In analogy to the semantical initialization, minimal nodes are initialized to *ff* and maximal nodes are initialized to *tt*. Disjunction, conjunction and first-order quantification are modeled by the respective formula operators. It remains to define the next-step computations. Starting with a finite first-order Kripke structure, we will have only finitely many successors to each (s, n). Let $n = (\mathbf{QX}\phi, \mathtt{W})$ and $(s, n) \xrightarrow{\mathtt{b},\mathtt{a}}{}' (s', n')$, where $a = \{\mathtt{v}_{i_1} \mathtt{:=?}, \ldots \mathtt{v}_{i_k} \mathtt{:=?}, \mathtt{v}_{j_1} \mathtt{:=t}_{j_1}, \ldots \mathtt{v}_{j_l} \mathtt{:=t}_{j_l}\}$, and let *fo-ann*$(s', n')$ be a first-order description of *ann*(s', n'). Then

$$p' =_{\text{df}} \begin{cases} \neg \mathtt{b} \vee (\forall \mathtt{v}_{i_1}, \ldots, \mathtt{v}_{i_k}. \textit{fo-ann}(s', n'))\ [\mathtt{t}_{j_1}/\mathtt{v}_{j_1}, \ldots, \mathtt{t}_{j_l}/\mathtt{v}_{j_l}] & \text{for } \mathbf{Q} = \mathbf{A} \\ \mathtt{b} \wedge (\exists \mathtt{v}_{i_1}, \ldots, \mathtt{v}_{i_k}. \textit{fo-ann}(s', n'))\ [\mathtt{t}_{j_1}/\mathtt{v}_{j_1}, \ldots, \mathtt{t}_{j_l}/\mathtt{v}_{j_l}] & \text{for } \mathbf{Q} = \mathbf{E} \end{cases}$$

describes the contribution of that successor to *fo-ann*$'(s, n)$. Now, if $\mathbf{Q} = \mathbf{A}$ then *fo-ann*$'(s, n)$ is the conjunction of all p' and if $\mathbf{Q} = \mathbf{E}$ then *fo-ann*$'(s, n)$ is the disjunction of all p'. The quantification over the input values stored in the variables $\mathtt{v}_{i_m}$ enables us, differing from [5], to cope with verification tasks where unboundedly many inputs are relevant.

The first-order description provides us with a syntactic procedure for first-order model checking that performs the same steps as the semantical process, except that termination may be harder to detect. To detect stabilization, we permit any kind of formula rewriting. If syntactic equality was the criterion for stabilization, a syntactic procedure would not even terminate for trivial problem instances. On the other hand, capturing propositional reasoning already seems to be sufficient for interesting applications. This is the case in our example.

Proposition 3. *If the syntactic model checking terminates then $K \models \phi$ if and only if for every initial state $(s, \mathtt{b}) \in I$, the condition $\mathtt{b}$ implies fo-ann$(s, (\phi, \emptyset))$.*

Note that, Proposition 3 is independent of any particular interpretation $\mathcal{I}$. We call the generated result of the syntactic model checking for K and ϕ to be the *verification condition*, which is

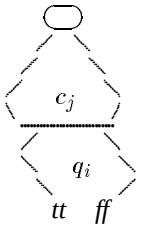
$$vc(K, \phi) =_{\text{df}} \bigwedge_{(s,\mathtt{b}) \in I} \mathtt{b} \Rightarrow \textit{fo-ann}(s, (\phi, \emptyset)).$$

Thus, $K \models \phi$ iff $\models vc(K, \phi)$. Termination of the syntactic annotation process is difficult to characterize. It depends on properties of the cycles in the product structure. In case of a data-independent system, the procedure can easily be seen to behave well, but there are far more interesting examples. If a cycle does not contain input, [5] provides a sufficient criterion. Even inputs are not harmful if tests do not interfere unrestricted, as for instance our running example shows. A more thorough treatment of this question must be deferred to a full version of this paper.

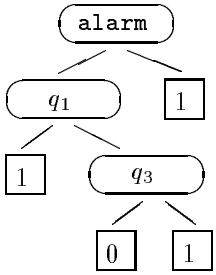
4 First-Order Model Checking, BDD-Based

First-order explicit model checking is time-consuming and only partly automatic. Thus the syntactic model-checking procedure above would not be terribly efficient if implemented. Our approach is based on the idea of combining BDD procedures for finite (small) data domains with first-order representations for infinite or large data domains. We assume that the variables V of the Kripke structure and quantified variables W from the FO-CTL specification are split accordingly into two classes: *Control* variables C whose valuations will be represented by BDDs, and *data* variables D whose valuations will be represented by first-order predicates.

First-order data predicates enter the BDD representation of an annotation in the form of propositions, which are atomic for the BDD. I.e., the BDD refers via proposition names to first-order formulas. Thus, one part of an annotation BDD B represents sets of control valuations, as an ordinary BDD would do. The rest describes what is left of *fo-ann*(s, n) after partial evaluation w.r.t. control values : first-order formulas, which are viewed as boolean combinations of data propositions. The meaning of the proposition names are kept in a separate *proposition table*. A more detailed description is given in the following subsection.



To the left, the general BDD form with control variables and proposition variables is depicted. A specific example BDD is shown to the right. It represents the first-order formula $q_1 \rightarrow (q_3 \vee \texttt{alarm=tt})$ that is the annotation of node $s_2 : \phi_{2,1} \rightarrow \phi_{2,2}$ in our example below.



4.1 Representing and Manipulating Data-Dependent Formulas

The proposition table includes, for each proposition variable q_i, the syntactical description of the first-order formula it represents and the set of data variables appearing free in that formula. Note that a proposition does not contain any control variables. The syntax of a proposition may include propositions which are defined earlier in the table. We say that a set of propositions *covers* a first-order formula, if the formula can be written as a boolean combination of these.

Operations : Assignments to be performed along transitions are reflected during model checking by changes to the annotations *fo-ann*(s, n). The effect of taking a transition on control variables is captured canonically using (functional) BDDs.

Assignments of data terms affect the annotations and hence the proposition table. Substitutions can be distributed to single propositions. They may lead to the introduction of new entries in the proposition table, if the result is not already covered by the present set. In our example (see Section 4.3) we apply the data assignment `k:=l` and the random assignment `l:=?` to a proposition q which represents `l-k>80`. Substituting `l'` for `l` and `l` for `k` yields the new proposition $q' \equiv$ `l'-l>80`. Quantifications have to be applied to whole boolean combinations, and they may enforce new propositions, too, e.g. $q'' \equiv \forall$`l'.l'-l>80`. Technically, we maintain a *substitution table* to manage the transformation which is induced on propositions by data assignments. E.g. it would contain q''/q to represent the substitution and quantification effect on q.

Restriction : separate control and data terms We will restrict ourselves to the simple case of a restricted interaction between control and data. This restriction concerns terms in the first-order Kripke structure and specification, but not the dynamically generated formulas *fo-ann*(s, n). We say that control and data are *separated*, if

- right-hand sides of assignments to data variables depend only on data variables,
- right-hand sides of assignments to control variables, conditions and boolean subformulas of the specification are *control terms*, which are generated by the rules :

$$\texttt{ctrl} ::= A \in \mathcal{A} \mid c \in \texttt{C} \mid \texttt{b(D)} \mid \texttt{f(ctrl,...,ctrl)}$$

Intuitively, this means that we may derive a boolean value from data to influence control. Similarly, control can of course influence data via conditions in the first-order structure which enable or disable transitions. The restriction ensures that we are able to represent formulas that are generated during the syntactic annotation process as described above. We conjecture that it is not strictly necessary to impose that restriction. Removing it would complicate our procedure, while its practical merits are debatable.

4.2 Execution

In this section we follow the syntactic model-checking procedure described in Section 3.3, except that now the annotations *fo-ann*(s, n) are represented by BDDs and the annotating process is done partially symbolically.

Initialization: As described in Section 3.3, terminal, minimal and maximal nodes of the product structure are initialized. Node annotations that contain pure control terms (without data variables) are stored as BDDs. In contrast, data dependencies are introduced into the proposition table together with a name q and a link to a single BDD node. This node represents the data proposition in the annotation.

Executing a step: If the annotation *fo-ann*(s, n) that has to be computed for some node (s, n) is a boolean combination of (already represented) subformulas then the representation of *fo-ann*(s, n) is the result of standard operations on BDDs.

Next-step computations of the model checker can be represented as purely boolean operations if no extensions to the proposition table are necessary and no quantifications have to be performed. Otherwise, we can carry out the substitution step symbolically, by enriching the proposition set on demand.

To compute the contribution of a transition $s \xrightarrow{\mathtt{b},\mathtt{a}} s'$ in an **AX** or **EX** evaluation we proceed as follows: we check whether the substitution table is defined for all propositions occurring in the BDD for the successor state. If not, we extend the proposition and substitution table accordingly. Also, there have to be propositions covering the boolean data terms in `b` and in assignments to control variables. The control transformations are captured in a second BDD, which has to be computed only once for the action.

Then, we compute the effect of the assignments as in a conventional symbolic model checker. If random assignments to data do occur in `a`, we have to perform the quantification afterwards. To that end, we extend, if necessary, the proposition table by new propositions with quantified variables and replace the affected parts in the BDDs by the new proposition variables. Finally, we can add the effect of the condition `b`. As a result, we get a BDD which represents the first-order formula *fo-ann*(s, n) from the syntactic model-checking procedure.

4.3 Application to the Example

In the running example, the only control variable is `alarm`, whereas `k`, `l` and `x` are data. The proposition table is initialized by q_1, q_2 and q_3 while the other rows in the table below are filled during a run of the algorithm. Starting the model checking bottom-up with the atomic propositions yields the annotation `ff` for node s_1`:in` and therefore — after some steps — the annotation `tt` for the left subtableau node s_1:ϕ_1. This annotation can be found quickly by performing control pruning as described in Section 5.

The right side of the product structure (on page 6) needs some more attention. We initialize the atoms s_2 `: in` with `tt`, s_2 `: l-x` $>$ `100` with q_1 and s_1 `: alarm=tt` with

`alarm=tt`. The annotation of node $s_2 : \phi_{2,2}$ is computed according to the definition in Section 3.3 along both transitions $s_2 \to s_1$. The first transition yields $\neg q_3 \vee$ `tt=tt` and the second one yields $q_3 \vee$ `alarm=tt`. Their conjunction simplifies to $q_3 \vee$ `alarm=tt`. Quantification is not necessary here, because both assignments are regular. Thus we get for the node $s_2 : \phi_{2,1} \to \phi_{2,2}$ the annotation $q_1 \to (q_3 \vee$ `alarm=tt`$)$.

name	proposition	data	var
q_1	`l-x>100`	`l`	
q_2	`l=x`	`l`	
q_3	`l-k>80`	`l,k`	
q_4	q_1`[l'/l] = l'-x>100`	`l'`	
q_5	q_3`[(l',l)/(l,k)] = l'-l>80`	`l',l`	

name	proposition	data	var
q_6	$\forall$ `l'. (`$q_4 \to q_5$`)`	`l`	
q_7	q_2`[l''/l'] = l''=x`	`l''`	
q_8	q_6`[l''/l]`	`l''`	
q_9	$\forall$ `l''. (`$q_7 \to q_8$`)`		
q_{10}	$\forall$ `x.` q_9		

In the next step of the algorithm the transition $s_1 \to s_2$ is used to manipulate this formula. `alarm` is replaced by `ff`. q_1 and q_3 depend on l that is substituted by the fresh variable l' while k is substituted by l. Afterwards l' is quantified. Note how we omit name clashes when substituting and quantifying l. The generated terms are stored in the proposition table, the substitutions are stored in the substitution table. Finally this step yields q_6 as annotation for node $s_1 : \mathbf{AX}\,(\phi_{2,1} \to \phi_{2,2})$. The formula q_6 does not change in the next bottom-up step of the model checking algorithm and also annotates $s_2 : \phi_{1,2}$. This yields $q_2 \to q_6$ for the node $s_2 : \phi_1$.

When we start the fixed point computation with annotations `tt` and $q_2 \to q_6$ and then step through both nodes in the loop we reach after some steps the stable description $(q_2 \to q_6) \wedge q_9$ for node $s_2 : \mathbf{AG}\,\phi_1$ and q_9 for node $s_1 : \mathbf{AG}\,\phi_1$.

The last step computes $\forall x.\, q_9$ as proof obligation for node $s : \phi$. Slightly simplified, this obligation states $\forall x, l.\, l - x > 100 \to l - x > 80$. Since this formula is a tautology, the first-order Kripke structure fulfills the specification.

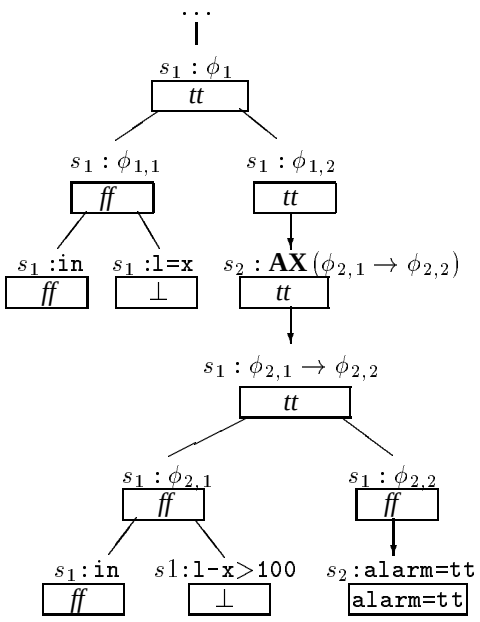
5 Optimization : Control Pruning

Control pruning is an optimization method that allows us to reduce the size of the product structure before the annotation procedure is applied speeding up this procedure. To this end, we will modify the procedure of computing annotations by permitting approximate computations for nodes, whose descendants are not yet stable.

In the simple version discussed here, we introduce a specific proposition "$\perp$" which replaces all data conditions. $\perp$ represents uncertainty. A BDD B depending on it yields a lower approximation of the correct annotation for $\perp =$ *ff* (i.e., B[*ff*/ $\perp$] implies the annotation), and an upper approximation for $\perp =$ *tt*. In favorable cases, some annotations can be computed to a value not depending on $\perp$ even if some annotations below still contain it. This permits us to reduce the size of the product structure to be annotated in the accurate procedure, so it also helps to keep the number of propositions small. The process of eliminating all irrelevant nodes after the simple approximation computation where all data propositions are replaced by $\perp$ is called *control pruning*, because it uses only control information to simplify the verification problem.

The approximation of annotations of the left part of the product structure of our example (the part below node $s_1 : \mathbf{AG}\,\phi_1$ in the picture on page 6) is presented on the next page. The approximation proves that the annotations below node $s_1 : \phi_1$ need not be considered again during a detailed model checking process, because the annotation

of this node already yields *tt*.
Applying approximation to the tree below the node $s_2 : \phi_1$ (right side in the product structure on page 6) yields $\perp$ for this node. Thus no optimization is possible there.



6 Conclusion

The most attractive aspect of our procedure is its closeness to standard symbolic model checking. It is an extension which permits to treat a selected set of variables differently. In fact, the first-order properties can be viewed as abstractions of the data values which are precise wrt. analysed property. In contrast to other frameworks of abstraction, they are computed automatically. Furthermore, they are tailored to the specification to be checked and allow partial evaluation of control to reduce the analysis effort. The method is currently implemented on top of a standard BDD package at OFFIS and will be employed in the context of verification of hybrid ECUs in automotive applications.

References

1. S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. submitted for publication.
2. Bernholtz, O., Vardi, M. and Wolper, P. *An automata-theoretic approach to branching-time model checking*, CAV '94, LNCS 818, 142–155.
3. Burch, J.R. and Dill, D.L. *Automatic verification of pipelined microprocessor control*, CAV '94, LNCS 818, 68–80.
4. Clarke, E.M., Grumberg, O. and Long, D.E. *Model checking and abstraction*, ACM TOPLAS, vol. 16,5. 1512–1542. 1994.
5. Damm W., Hungar H., and Grumberg O. *What if model checking must be truly symbolic*. LNCS 987, 1995.
6. Dingel, J. and Filkorn, T. *Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving*, CAV '95, LNCS 939, 1995
7. Graf, S. *Verification of a distributed cache memory by using abstractions*, CAV '94, LNCS 818 (1994), 207–219.
8. A.J. Isles, R. Hojati, and R.K. Brayton. Computing reachability control states of systems modeled with uninterpreted functions and infinite memory. In *Proc. of CAV'98*, LNCS.
9. K. Sajid, A. Goel, H. Zhou, A. Aziz, S. Barber, and V. Singhal. BDD based procedures for the theory of equality with uninterpreted functions. In *Proc. of CAV'98*, LNCS, 1998.
10. Wolper, P. *Expressing interesting properties of programs in propositional temporal logic*, POPL '86, 184–193.
11. Xu Y., Cerny E., Song X., Corella F., and Mohamed O.A.. *Model checking for a first-order temporal logic using multiway decision graphs*. In *Proc. of CAV'98*, LNCS, 1998.