Combining Constraint Solving and Symbolic Model Checking for a Class of Systems with Non-linear Constraints

William Chan*, Richard Anderson, Paul Beame, David Notkin

Computer Science and Engineering, University of Washington, Box 352350
Seattle, WA 98195-2350, U.S.A.
{wchan,anderson,beame,notkin}@cs.washington.edu

Abstract. We extend the conventional BDD-based model checking algorithms to verify systems with non-linear arithmetic constraints. We represent each constraint as a BDD variable, using the information from a constraint solver to prune the BDDs by removing paths that correspond to infeasible constraints. We illustrate our technique with a simple example, which has been analyzed with our prototype implementation.

1 Introduction

Although symbolic model checking [BCM+90] based on Binary Decision Diagrams [Bry86], or BDDs, has been remarkably successful for verifying finite state systems, it fails when complex arithmetic constraints are present. For example, if the bits of the integers x, y and z are represented as BDD variables, the BDD for the non-linear constraint $xy = z$ has exponential size [LS81]. In this paper, we tightly couple a constraint solver with a BDD-based model checker to verify systems with possibly non-linear arithmetic constraints.

A large class of embedded, reactive systems consist of a finite-state *control* component together with *numeric data inputs* that measure quantities such as velocity, temperature, etc. In these systems, state transitions depend on predicates, or *constraints*, on these numerical values.

We have been studying the practicality of model checking for specifications of large and complex reactive software systems of this type. Our major effort has been directed at the preliminary requirements of one such system, TCAS II, an airborne collision avoidance system used on many commercial aircraft. In [ABB+96] we applied BDD-based model checking to about one third of the TCAS II specification, discovering a number of violations of desirable properties. The full specification—expressed in RSML [LHHR94], a dialect of Statecharts [Har87]—comprises about 400 pages.

Our approach for handling constraints exploited finiteness of the data input domains, representing each bit of data input as a BDD variable and constraints by BDDs in these variables. This worked well when dealing with purely linear

* Supported by a Microsoft Graduate Fellowship

constraints but did not extend efficiently to non-linear constraints, such as those found in the remaining portions of TCAS II.

In this paper, we propose to represent each (linear or non-linear) constraint, instead of each bit, as a BDD variable. For soundness and completeness, infeasible combinations of constraints have to be detected, which we do using an auxiliary constraint solver.

The class of systems we consider is defined by a restriction on the updates to data values: a transition must either set all new data values based only on absolute properties of their current values, or else leave them unchanged. A key property of such systems is that the decision to take a transition depends on the current data only via Boolean combinations of the constraints originally present in the specification. This restriction was also partly motivated by the semantics of RSML and, although it cannot handle all of TCAS II, it does allow modeling of a significant portion of it. We define our system model and show the key property of the restrictions on the transitions in Sect. 2.

Given the key property, a simple approach to combining the model checker and constraint solver is to test all combinations of constraints for feasibility before applying model checking. We develop a potentially more efficient approach whereby we prune the infeasible paths from the BDDs on the fly. We present our model checking algorithms in Sect. 3. and give a simple example that has been analyzed with our prototype implementation in Sect. 4.

Related Work. We have opted to augment BDD-based model checking to deal with non-linear constraints. The main reason is that we are interested in systems with large and complex control logic, for which only BDD-based model checking has proven to work well. The high dependence between control and data paths also prevents us from separating them for verification, a technique that is sometimes used in microprocessor verification.

Most work on handling non-linearity in verification has been focused on arithmetic circuits. One approach is to use BMDs or *BMDs [BC95] and their variants, such as HDDs [CFZ95]. Although they can represent the product xy concisely, representing the constraint $xy = z$ still requires exponential size. In fact, Thathachar [Tha96] shows that small variations of these representations are not likely to solve the problem. Our approach can deal with not only integral multiplicative constraints but also arbitrarily complex (e.g. trigonometric) constraints over finite or infinite domains, provided an appropriate constraint solver is available.

Abstracting a constraint as a single Boolean variable is not a new idea (e.g., [CDV96]). However, since infeasible combinations of constraints are not automatically detected, either the approach is incomplete for safety properties, or it requires substantial manual abstraction. Wang et al. [WME93] also represent certain timing constraints in distributed real-time systems as BDD variables. However, to ensure soundness and completeness, their method requires building a BDD in exponential time before running the fixed-point algorithm. We try to avoid a similar preprocessing by restricting the class of systems that we deal with and by filtering the BDDs on the fly.

Note that the work on nonlinear hybrid systems [HH95] differs from ours since it is concerned with constraints that are non-linear *differential equations.*

2 Models

We first give the definitions of basic transition systems, bisimulation equivalence, and quotient systems. Then we present our system model, whose semantics can be defined in terms of a basic transition system, and then show that certain restrictions on the transitions give rise to a natural bisimulation.

2.1 Basic Transition Systems

A reactive system can be modeled as a *basic transition system* $\langle Q, Q_0, \rightarrow, \Sigma, L\rangle$, where Q is a set of states, $Q_0 \subseteq Q$ is a set of initial states, $\rightarrow \subseteq Q \times Q$ is the transition relation, Σ is a set of atomic propositions, and $L: Q \mapsto 2^{\Sigma}$ labels each state with the set of atomic propositions in Σ that are true in that state. If we have $q \rightarrow q'$, then the state q' is called a *successor* of q.

Intuitively, an observer sees the label of the current state, but not the state itself. Two states are indistinguishable if their labels are the same and their successors are again indistinguishable. Formally, we say that an equivalence relation $\approx$ of Q is a *bisimulation* (cf. [Mil80, pp. 42]) if for all states q_1 and q_2, we have that $q_1 \approx q_2$ implies (1) $L(q_1)$ equals $L(q_2)$ and (2) for all q_1' in Q with $q_1 \rightarrow q_1'$, there exists a q_2' in Q with $q_2 \rightarrow q_2'$ and $q_1' \approx q_2'$.

The quotient system of $\langle Q, Q_0, \rightarrow, \Sigma, L\rangle$ with respect to a bisimulation $\approx$ is a basic transition system $\langle Q^{\approx}, Q_0^{\approx}, \rightarrow^{\approx}, \Sigma, L^{\approx}\rangle$. The quotient state space $Q^{\approx}$ is the set of equivalence classes induced by $\approx$. For all S and S' in $Q^{\approx}$, we have $S \rightarrow^{\approx} S'$ if and only if there exist an s in S and an s' in S' with $s \rightarrow s'$. We define $L^{\approx}(S) = L(s)$ for any s in S, and $Q_0^{\approx} = \{S \in Q^{\approx} \mid S \cap Q_0 \neq \emptyset\}$. We say that $\approx$ is *finite* if $Q^{\approx}$ is finite.

Many properties of $\langle Q, Q_0, \rightarrow, \Sigma, L\rangle$ can be expressed in the temporal logic CTL* [EH86] as formulas whose atomic propositions are taken from Σ. CTL* is strictly more expressive than CTL and LTL, commonly used in model checking. For our methods we need the following theorem (see, for example, [BCG88] for a proof of a similar theorem):

Theorem 1. *Any CTL* formula f is true in a basic transition system M if and only if f is true in the quotient system of M with respect to any bisimulation.*

2.2 System Model

We are interested in reactive systems with a finite control component and a finite or infinite numeric data component. The control component is represented by a finite set N of *control nodes*. The data component is represented by a finite vector $\mathbf{x}$ of *data variables*, and the domain of each variable is a finite or infinite subset of $\mathbb{R}$, the set of reals. Let D be the Cartesian product of the domains of

the data variables. An assignment to $\mathbf{x}$ denotes a point in D, and a *constraint* on $\mathbf{x}$ denotes a subset of D. More explicitly, a constraint $c(\mathbf{x})$ is a predicate of the form $g(\mathbf{x}) \bowtie 0$ with $g: D \mapsto \mathbb{R}$ and $\bowtie$ is one of $\{<, \leq, =, \neq, \geq, >\}$. If we have $g(\mathbf{x}) \equiv \mathbf{a} \cdot \mathbf{x} + b$ for some vector $\mathbf{a}$ and constant b, then the constraint is *linear*. We are interested in both linear and non-linear constraints. We also call any finite Boolean combination of constraints a constraint. We denote by $[\![c(\mathbf{x})]\!]$ the set of points in D that satisfy $c(\mathbf{x})$. The constraint $c(\mathbf{x})$ is *feasible* if and only if $[\![c(\mathbf{x})]\!]$ is not empty. We write c for $c(\mathbf{x})$ when there is no ambiguity.

Our *system model* is a tuple $\langle N, N_0, \mathbf{x}, D, \Delta, C\rangle$, where N, $\mathbf{x}$, and D are defined as above, $N_0 \subseteq N$ is a set of initial control nodes, Δ is a mapping from N^2 to $2^{D\times D}$, and C is a finite set of constraints on $\mathbf{x}$. The system model defines a basic transition system $\langle Q, Q_0, \rightarrow, \Sigma, L\rangle$ as follows. The state space Q is $N \times D$. The set of initial states Q_0 is $N_0 \times D$. We define $L(v,a) = \{v\} \cup \{c \in C \mid a \in [\![c]\!]\}$ and $\Sigma = N \cup C$. Intuitively, this choice of labeling implies that the control nodes are fully observable, while data points are only distinguishable through the constraints in C.

The transition relation $\rightarrow$ is defined so that for all (v,a) and (v',a') in $N \times D$, $(v,a) \rightarrow (v',a')$ if and only if (a,a') is in $\Delta(v,v')$. If we define $\mathbf{x}' = (x_1', x_2', \ldots, x_m')$, the "next-state" version of $\mathbf{x} = (x_1, x_2, \ldots, x_m)$, then we can think of Δ as specifying as a mapping from pairs of nodes to joint constraints on $\mathbf{x}$ and $\mathbf{x}'$. That is, for any v and v' in N, we have $\Delta(v,v') = [\![\alpha(\mathbf{x},\mathbf{x}')]\!]$ for some constraint $\alpha(\mathbf{x},\mathbf{x}')$. For example, if $\Delta(v,v')$ is $[\![x_1 > 0 \ \wedge \ x_1' = x_1 + 1]\!]$ and the domain of x_1 is $\mathbb{R}$, then $(1,2) \in \Delta(v,v')$ so $(v,1) \rightarrow (v',2)$ is a possible transition.

2.3 Restrictions on Transitions

The system model defined above is very general and contains classes of systems that are undecidable or intractable for model checking. We restrict our attention to system models with the following property on Δ.

Property 2. *For all (v,v') in N^2, $\Delta(v,v')$ is either*

1. $[\![\alpha_1(\mathbf{x}) \ \wedge \ \alpha_2(\mathbf{x}')]\!]$, *or*
2. $[\![\alpha_1(\mathbf{x}) \ \wedge \ \alpha_2(\mathbf{x}') \ \wedge \ \mathbf{x}' = \mathbf{x}]\!]$

where $\alpha_1(\mathbf{x})$ and $\alpha_2(\mathbf{x})$ are some Boolean combinations of constraints from C.

In the above definition, $\alpha_2(\mathbf{x}')$ is the renaming of $\alpha_2(\mathbf{x})$ with the occurrences of $\mathbf{x}$ replaced by $\mathbf{x}'$. We call the first kind of transition above *data-memoryless*. The idea is that the value of $\mathbf{x}'$ is independent of $\mathbf{x}$. For example, $\Delta(v,v') = [\![x_1 < 3 \ \wedge \ x_1' > 5]\!]$ satisfies the property (if the constraints $x_1 < 3$ and $x_1 > 5$ are in C). The second kind of transition is called *data-invariant* since the values of all the data variables remain unchanged after the transition.

Property 2 may seem quite restrictive. Even the simple constraint $x_1' = x_1 + 1$ mentioned earlier is ruled out. However, it does allow complex "guarding conditions", like $x_1 x_2 < x_3$ or $x_1 > \sin x_2$, etc. As we will see in Sect. 4, this property

is naturally exhibited by certain Statecharts machines whose internal steps, while responding to particular changes in their environment, may be modeled as data-invariant transitions and whose environment may be modeled conservatively via data-memoryless transitions.

The key observation is that for any system model with the above property, the equivalence relation induced by the labeling is a bisimulation. Furthermore, the bisimulation is finite even if D is infinite.

Theorem 3. *Given a system model with state space $N \times D$ and labeling function L, let $\sim$ be the equivalence relation of $N \times D$ such that for all (v_1, a_1) and (v_2, a_2) in $N \times D$, we have $(v_1, a_1) \sim (v_2, a_2)$ if and only if $L(v_1, a_1)$ equals $L(v_2, a_2)$. The relation $\sim$ is a finite bisimulation for system models that satisfy Property 2.*

3 Model Checking

As a result of Theorems 1 and 3, given a system model with Property 2 and a CTL* formula, it is sufficient to verify the quotient system with respect to $\sim$. In this section, we first describe a Boolean encoding of the quotient system, and give a straightforward model checking algorithm which requires an exponential-time preprocessing stage to build a special BDD. Then we explain how that may be avoided by an operation we call filtering. Although the worst-case time complexity of filtering BDDs is also exponential, the hope is that the actual time required is less than the worst case.

We assume that we have a constraint solver that given a set of constraints can determine whether their conjunction is feasible. This problem has been studied by the constraint logic programming (CLP) community to extend CLP languages for non-linear constraints, and also by the operations research community to solve constrained optimization problems by first finding a feasible point.

3.1 A Boolean Encoding for Model Checking

Given a system model $\langle N, N_0, \mathbf{x}, D, \Delta, C \rangle$, the quotient state space with respect to $\sim$, i.e. the set of equivalence classes of $N \times D$ induced by $\sim$, is of the form $N \times D^{\sim}$ where $D^{\sim}$ is a collection of disjoint subsets of D, which we call *regions*, defined by the set of constraints in C that are true on those data points.

Our goal is to encode the quotient system symbolically by a set of Boolean variables so that BDDs can be used. The control part is encoded in a conventional manner: we encode the node $v \in N$ in some convenient way as an assignment $\psi_N(v)$ to a vector $\mathbf{v}$ of n Boolean variables with $n \geq \lceil \log |N| \rceil$, e.g. as the binary encoding of a number between 1 and $|N|$.

The way we handle the data part, $D^{\sim}$, distinguishes our approach from others. For $C = \{c_1, \ldots, c_m\}$, each region is of the form $[\![\alpha]\!]$ with $\alpha \equiv \bigwedge_{1 \leq i \leq m} l_i$ where l_i is either c_i or $\neg c_i$. This suggests a natural embedding ψ_D of $D^{\sim}$ into $\{0,1\}^m$ in which an assignment to a vector $\mathbf{k}$ of m Boolean variables $k_1, k_2, \ldots, k_m$ encodes a region $[\![\alpha]\!]$ if k_i is set to 1 exactly when c_i occurs positively in

α. We also define a Boolean function $Feas(\mathbf{k})$ such that $Feas(\overline{k}) = 1$ if only if $\overline{k} \in \mathrm{Im}\psi_D$, i.e. $\overline{k}$ encodes a feasible constraint.

A state in the quotient system is encoded as an assignment to $(\mathbf{v}, \mathbf{k})$, and a set of states can be represented in the standard way as a Boolean function $S(\mathbf{v}, \mathbf{k})$, such that a state $(\overline{v}, \overline{k})$ is in the set if and only if $S(\overline{v}, \overline{k}) = 1$. (As is usual, we will think of S as a function and as a set interchangeably.) In general an arbitrary S may contain *infeasible states* — assignments to $(\mathbf{v}, \mathbf{k})$ with $Feas(\mathbf{k}) = 0$ — that we can remove by computing $S \wedge Feas$.

We now define a transition relation R on $\{0,1\}^{n+m}$ that encodes the transitions of the quotient system on $N \times D^{\sim}$. That is, we define a Boolean function $R(\mathbf{v}, \mathbf{k}, \mathbf{v}', \mathbf{k}')$, where $\mathbf{k}' = (k'_1, k'_2, \ldots, k'_m)$ and $\mathbf{v}'$ are the next-state versions of $\mathbf{k}$ and $\mathbf{v}$ respectively, that represents the transition relation of the quotient system. A natural condition in doing this would be to restrict $R(\overline{v}, \overline{k}, \overline{v'}, \overline{k'})$ to be 1 only if $(\overline{v}, \overline{k})$ and $(\overline{v'}, \overline{k'})$ each encode elements of $N \times D^{\sim}$; however this may lead to a very large BDD for R if the BDD for $Feas$ is large. Instead, we permit R to be 1 for values of $\overline{k}$ and $\overline{k'}$ that encode infeasible constraints and rely on manipulation of the state representations to eliminate infeasible states.

More precisely, let $\Delta(v, v')$ be $[\![\alpha_{v,v'}(\mathbf{x}, \mathbf{x}')]\!]$ for some constraint $\alpha_{v,v'}(\mathbf{x}, \mathbf{x}')$ which satisfies Property 2. If we replace each $c_i(\mathbf{x})$ and $c_i(\mathbf{x}')$ in $\alpha_{v,v'}(\mathbf{x}, \mathbf{x}')$ with k_i and k'_i respectively (just as in our encoding of quotient states) and conjoin $k_i = k'_i$ for $i = 1, \ldots, m$ if the transition is data-invariant, we obtain a Boolean function $\chi_{v,v'}(\mathbf{k}, \mathbf{k}')$. It can be shown that if $(\overline{v}, \overline{k})$ and $(\overline{v'}, \overline{k'})$ encode states $(v, [\![\alpha]\!]), (v, [\![\alpha']\!]) \in N \times D^{\sim}$, then $(\overline{v'}, \overline{k'})$ is a successor of $(\overline{v}, \overline{k})$ if and only if $\chi_{v,v'}(\overline{k}, \overline{k'}) = 1$. The relation $R(\mathbf{v}, \mathbf{k}, \mathbf{v}', \mathbf{k}')$ is then

$$\bigvee_{(v,v') \in N^2} (\mathbf{v} = \psi_N(v) \wedge \mathbf{v}' = \psi_N(v') \wedge \chi_{v,v'}(\mathbf{k}, \mathbf{k}')) .$$

The BDD for R is easy to build from the system model description and thus conventional model checking algorithms can now be used to compute in the quotient system, provided that we also conjoin each set of states encountered with $Feas$ to remove infeasible states.

However, even if the BDD for $Feas$ is small, in general there may be no efficient way of computing it. The naive method enumerates all 2^m assignments to $\mathbf{k}$ and invokes the constraint solver to check the feasibility of each case. This method may work well if the number of constraints m is small.

3.2 Filtering

We can avoid building the BDD for $Feas$ if we have some other way of removing infeasible states. One solution is *filtering* the functions on the fly. We represent an arbitrary function S by a BDD in the implementation which, to simplify the terminology when explaining filtering, we think of as simply a DNF formula representing S, consisting of the disjunction of all the paths from the root to the leaf 1. The idea of filtering is that, instead of computing $S \wedge Feas$, we remove every disjunct d of S with $d \wedge Feas \equiv 0$. We denote the resulting function as $Filter_{Feas} S$. (Note that the value of $Filter_{Feas} S$ depends on the particular DNF representation for S.)

Since every disjunct d is a conjunction, we can determine whether d is feasible using the constraint solver, without computing $Feas(\mathbf{k})$. Note also that $Filter_{Feas}S$ and $S \wedge Feas$ are not necessarily the same function. For example, let S be the constant function 1, which can also be its DNF representation. Then, we have $S \wedge Feas \equiv Feas$ but $Filter_{Feas}S \equiv 1$. In general, we have $(S \wedge Feas) \subseteq Filter_{Feas}S \subseteq S$ (the inclusion is referring to the sets represented by the Boolean functions). Although $Filter_{Feas}S$ still contains some infeasible states, we will show that it is sufficient for model checking.

The algorithms for symbolic model checking [BCM+90] involve four types of operations on sets of states: Boolean operations, emptiness checking, image (or pre-image) computation, and finding elements in non-empty sets (for counterexample traces). The lemma below is easy to prove and implies that for Boolean operations we can delay the removal of infeasible states until the end (S and T are arbitrary Boolean functions).

Lemma 4. *We have the following equalities:*

(i) $(S \wedge Feas) \wedge (T \wedge Feas) \equiv (S \wedge T) \wedge Feas$.
(ii) $(S \wedge Feas) \vee (T \wedge Feas) \equiv (S \vee T) \wedge Feas$.
(iii) $(\neg(S \wedge Feas)) \wedge Feas \equiv (\neg S) \wedge Feas$.

The functions on the left hand side are the straightforward way of doing the operations. On the right hand side, we do the same operations but remove infeasible states only in the final result. The next lemma implies that if we only care whether the set is empty, then even the final result does not need to be intersected with $Feas$; instead, we can check the emptiness of the filtered result.

Lemma 5. $S \wedge Feas \equiv 0$ *if and only if* $Filter_{Feas}S \equiv 0$.

The next lemma gives a way of computing the image (i.e., successors) of a set of states without using $Feas$ (pre-image computation is similar).

Lemma 6. *We have the following equality:*

$$\begin{aligned}&\exists \mathbf{v}.\, \exists \mathbf{k}.\, (Feas(\mathbf{k}) \wedge S(\mathbf{v},\mathbf{k}) \wedge R(\mathbf{v},\mathbf{k},\mathbf{v}',\mathbf{k}'))\\ &\equiv \exists \mathbf{v}.\, \exists \mathbf{k}.\, \left(Filter_{Feas(\mathbf{k})}\,(S(\mathbf{v},\mathbf{k}) \wedge R(\mathbf{v},\mathbf{k},\mathbf{v}',\mathbf{k}'))\right).\end{aligned}$$

As a result of the above three lemmas, the only necessary change to the conventional symbolic model checking algorithms is to use the right hand sides of Lemmas 5 and 6 to detect convergence and compute images respectively. Finally, the following lemma implies a way of finding a feasible state in a set.

Lemma 7. *If we have* $Filter_{Feas}S \not\equiv 0$, *then for each disjunct* d *of* $Filter_{Feas}S$, *there exists an assignment to the input variables of* S *with* $d \wedge Feas = 1$.

So to find a feasible state in S, we compute $Filter_{Feas}S$ and pick an arbitrary disjunct d, which corresponds to a partial assignment to the variables. To get a complete assignment, the unassigned variables not in $\mathbf{k}$ can be set arbitrarily. For the unassigned variables in $\mathbf{k}$, we can set them one by one using information

```
FILTER(B: BDD): BDD
    LABEL(B,true)
    return PRUNE(B)

PRUNE(B: BDD): BDD
    if B = 0 or B = 1 then return B
    let y_j = B.Var
    if j > l then return B
    if ⟨B, B'⟩ is in cache, return B'
    if B.Ledge = ⊤
        then B_0 ← PRUNE(B.Lchild)
        else B_0 ← 0
    if B.Redge = ⊤
        then B_1 ← PRUNE(B.Rchild)
        else B_1 ← 0
    B' ← ITE-BDD(B.Var, B_0, B_1)
    insert ⟨B, B'⟩ and ⟨B', B'⟩ in cache
    return B'

LABEL(B: BDD, α: Constraint): {⊤,⊥}
    if B = 0 then return ⊥
    if B = 1 then return FEAS(α)
    let y_j = B.Var
    case
        j < u: ..... (case 1: upper layer)
            if B.Ledge = ? then
                r_0 ← LABEL(B.Lchild, α)
                B.Ledge ← r_0
            else r_0 ← B.Ledge
            if B.Redge = ? then
                r_1 ← LABEL(B.Rchild, α)
                B.Redge ← r_1
            else r_1 ← B.Redge
        u ≤ j ≤ l: ... (case 2: middle layer)
            r_0 ← LABEL(B.Lchild, α ∧ I(¬y_j))
            if r_0 = ⊤ then B.Ledge ← ⊤
            r_1 ← LABEL(B.Rchild, α ∧ I(y_j))
            if r_1 = ⊤ then B.Redge ← ⊤
        j > l: ..... (case 3: lower layer)
            return FEAS(α)
    endcase
    if r_0 = ⊤ or r_1 = ⊤ then return ⊤
    else return ⊥
```

Fig. 1. A BDD filtering algorithm

from the constraint solver: pick an unassigned variable and arbitrarily set it to 0, and if the extended assignment is not feasible, revert it to 1 (the new extended assignment is guaranteed to be feasible). Repeat until all the variables are set.

3.3 Filtering BDDs

Filtering a BDD amounts to removing all paths from the root to the leaf 1 that correspond to infeasible constraints. Figure 1 shows a BDD filtering algorithm FILTER. We assume that the given BDD is a function of $\mathbf{v}$ and $\mathbf{k}$, which is being filtered with respect to $Feas(\mathbf{k})$. (What we will describe can be easily generalized to handle functions of $(\mathbf{v}, \mathbf{k}, \mathbf{v}', \mathbf{k}')$ and filtering with respect to $Feas(\mathbf{k}')$.) The algorithm consists of two phases: in the labeling phase, it labels the edges along all feasible paths with $\top$, and in the pruning phase, it redirects the edges not labeled with $\top$ to the leaf 0.

Each non-leaf BDD node has five fields. The *Var* field stores the BDD variable. The *Lchild* field points to the 0-child BDD. The *Ledge* field is the label of the left edge, which is either $\top$ (feasible), $\bot$ (infeasible), or ? (unknown, the initial value). The *Rchild* and *Redge* fields are symmetric. Suppose the BDD variables in order are $y_1, y_2, \ldots, y_u, \ldots, y_l, \ldots, y_{m+n}$, where y_u and y_l are the

first and last variables in $\mathbf{k}$. We call the part of the BDD with variables y_1 through y_{u-1} the upper layer, y_u through y_l the middle layer, and y_{l+1} through y_{m+n} the lower layer. Therefore, only the middle layer contains variables in $\mathbf{k}$.

The routine LABEL traverses the paths in a depth-first manner, keeping track of the corresponding constraint α as it walks down a path. Case 2 is important for correctness, while cases 1 and 3 are for optimizations—each node in the upper layer is not visited more than once (case 1), and nodes in the lower layer are not explored at all (case 3). The constraint solver $\mathcal{FEAS}$ takes a constraint α, and returns $\top$ if α is feasible, or $\bot$ otherwise. The function $\mathcal{I}$ "interprets" the BDD variables as data constraints. For each v_i in $\mathbf{v}$, we have $\mathcal{I}(\neg v_i) = \mathcal{I}(v_i) = \mathit{true}$, and for each k_i in $\mathbf{k}$, we have $\mathcal{I}(k_i) = c_i$ and $\mathcal{I}(\neg k_i) = \neg c_i$. The routine PRUNE performs the pruning phase. The function *ITE-BDD* takes a BDD variable y and two BDDs B_0 and B_1, and returns a BDD with top variable y, 0-child B_0 and 1-child B_1.

Assuming that $\mathcal{FEAS}$ takes constant time, the time complexity of FILTER is linear in the number of nodes in the upper layer, and in the number of *paths* in the middle layer (which is the major bottleneck of the algorithm).

A Refinement. It makes sense to filter the BDDs instead of building *Feas* only if the number of paths checked is smaller than 2^m. Unfortunately, filtering $S \wedge R$ as suggested by Lemma 6 may be very expensive. To see this let $R(\mathbf{v}, \mathbf{k}, \mathbf{v}', \mathbf{k}')$ be $R_1(\mathbf{v}, \mathbf{k}, \mathbf{v}', \mathbf{k}') \vee (R_2(\mathbf{v}, \mathbf{k}, \mathbf{v}', \mathbf{k}') \wedge \mathbf{k}' = \mathbf{k})$, where R_1 and $R_2 \wedge \mathbf{k}' = \mathbf{k}$ represent the data-memoryless and data-invariant transitions respectively. The constraint $\mathbf{k}' = \mathbf{k}$ conjoined with R_2 introduces a path for each possible assignment to $\mathbf{k}$, so there may be 2^m paths to check. However, the observation is that we can rename each k_i' in R_2 to k_i without changing the function $R_2 \wedge \mathbf{k}' = \mathbf{k}$, thus eliminating $\mathbf{k}'$ from R_2. We have the following lemma.

Lemma 8. *The following equality holds:*

$$\begin{aligned} &\left(\exists \mathbf{v}.\, \exists \mathbf{k}.\, \left(\mathit{Filter}_{\mathit{Feas}(\mathbf{k})}\left(S(\mathbf{v}, \mathbf{k}) \wedge R(\mathbf{v}, \mathbf{k}, \mathbf{v}', \mathbf{k}')\right)\right)\right) \wedge \mathit{Feas}(\mathbf{k}') \\ \equiv\ &(U(\mathbf{v}', \mathbf{k}') \vee V(\mathbf{v}', \mathbf{k}')) \wedge \mathit{Feas}(\mathbf{k}') \end{aligned}$$

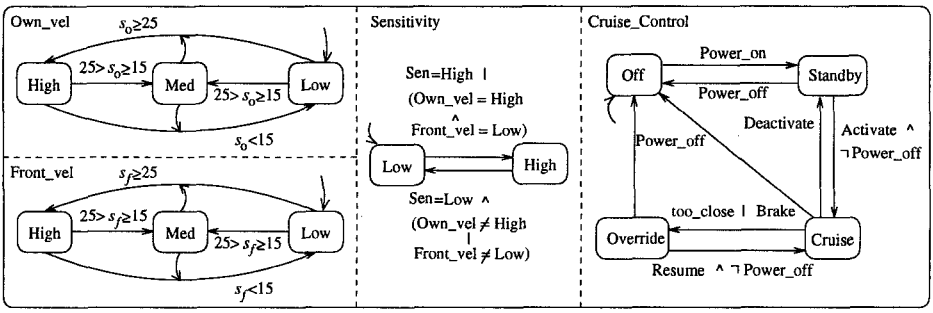
with

$$\begin{aligned} U(\mathbf{v}', \mathbf{k}') &= \exists \mathbf{v}.\, \exists \mathbf{k}.\, \mathit{Filter}_{\mathit{Feas}(\mathbf{k})}\left(S(\mathbf{v}, \mathbf{k}) \wedge R_1(\mathbf{v}, \mathbf{k}, \mathbf{v}', \mathbf{k}')\right) \\ V(\mathbf{v}', \mathbf{k}) &= \exists \mathbf{v}.\, \mathit{Filter}_{\mathit{Feas}(\mathbf{k})}\left(S(\mathbf{v}, \mathbf{k}) \wedge R_2(\mathbf{v}, \mathbf{k}, \mathbf{v}')\right). \end{aligned}$$

So we compute $U \vee V$, handling the constraint $\mathbf{k} = \mathbf{k}'$ implicitly.

4 Implementation and Example

We implemented the above algorithms in SMV [McM93]. The constraint solver used was QUAD-CLP(IR) [PB94], a less incomplete solver than CLP(IR) for quadratic constraints. We had access only to the executable of the solver, so it was integrated with SMV through interprocess communication.



Inputs	
$s_o \in \mathbb{R}$	Power_On : boolean
$s_f \in \mathbb{R}$	Power_Off : boolean
$d \in \mathbb{R}$, $d \geq 0$	Activate : boolean
Sen : {High, Low}	Deactivate : boolean
Brake : boolean	Resume : boolean

Abbreviation

too_close $\equiv \frac{d - K/d}{\max(1,s)} < 2$,
where $s = s_o - s_f$ and

$$K = \begin{cases} 1 & \text{when Sensitivity = Low} \\ 10 & \text{when Sensitivity = High} \end{cases}$$

Fig. 2. A hypothetical automobile cruise control system with collision avoidance

Because a motivation of this work is to analyze the TCAS II requirements, we illustrate our technique with a simple Statecharts-like system shown in Fig. 2. It is a hypothetical automobile cruise control system with collision avoidance. The idea is that when the automobile is too close to the vehicle in front, the cruise control system will automatically deactivate itself. (In addition to TCAS, the example was influenced by the one used by Atlee and Gannon [AG93].)

Three inputs to the system are s_o, the velocity of the vehicle; s_f, the velocity of the front vehicle; and $d \geq 0$, the distance between the vehicles. (In reality, s_f may be estimated from the current and previous values of s_o and d.) The closeness of the two vehicles is based on time rather than distance. Let s be $s_o - s_f$. The estimated time to collision is d/s. If this quantity is less than some threshold t, the two vehicles may be considered too close. However, if s is positive but small, the two vehicles can get very close without triggering the condition. To fix this problem, the following condition can be used instead:

$$\frac{d - K/d}{\max(\epsilon, s)} < t.$$

The max function is for avoiding division-by-zero. Subtracting K/d from the numerator makes the inequality true when d is tiny, regardless of the value of s. The positive value K depends on the "sensitivity level" (large K for high sensitivity). Although this example is naive, the inequality is exactly the one used in TCAS II for threat detection. We arbitrarily chose $\epsilon = 1$ and $t = 2$.

As shown in Fig. 2, the system is divided into four components. In Statecharts and RSML, different components are synchronized by *events* (signals). We omit

this mechanism in the figure and assume the components execute in the following order: in the first *micro-step*, Own_vel and Front_vel execute concurrently (i.e., each takes one enabled transition, or stays unchanged if none is enabled); in the second and third micro-steps, Sensitivity and Cruise_Control execute respectively. The three micro-steps together form a *super-step*. Transitions in a component are guarded by assertions on the inputs and/or other components. It should be clear that we can construct the product system of the components in the usual manner and represent it with a system model (Sect. 2.2). The values of all the inputs are nondeterministic at the beginning of a super-step, but during a super-step they are assumed to be unchanged by the so-called *synchrony hypothesis*. Therefore this system satisfies Property 2: micro-step transitions are data-invariant, while transitions across super-steps are data-memoryless.

We verified several safety properties of a model of this system using our prototype implementation. In the model, there are (at least) six Boolean variables representing constraints: $s_o \geq 25, s_o < 15, s_f \geq 25, s_f < 15, ((d - 1/d)/\max(1, s_o - s_f)) < 2, ((d - 10/d)/\max(1, s_o - s_f)) < 2$. Additional Boolean variables are used when the property being verified contains other constraints. We focused on verifying that Cruise_Control is never in the Cruise node under certain conditions, for example, when d is less than 2, (That is, in CTL, AG $!(d < 2$ & Cruise_Control = Cruise).) This property is false because the two transitions into the Cruise node are not guarded by ¬too_close. The model checker correctly showed a counterexample. After strengthening the guards on the two transitions, the property was verified true. Two other related properties were also successfully verified: Cruise_Control is never in the Cruise node when either (1) d is less than 4 and Sensitivity is High, or (2) d is less than 20, Sensitivity is High, and Sen is Low. Each of the above specifications was evaluated within a second by our prototype implementation. The numbers of calls made to the constraint solver were at most about 30% of the number of calls required to construct *Feas*.

5 Conclusion

The technique described in this paper can be generalized in various ways. The idea can be applied to systems with transitions annotated by assertions in any theory, if a decision procedure for the theory is available. Allowing transitions that are not data-memoryless or data-invariant can make the technique more useful, but that would probably require computing a bisimulation before applying model checking, or approximating one on the fly. Doing so may blow up the number of BDD variables. There are also many open questions for the short term. We need to experiment with larger systems like TCAS II to see whether the technique is practical. The choice of variable ordering also needs investigation because it affects both the BDD size and the number of paths traversed.

References

[ABB+96] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. In *Pro-*

ceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 156–166, October 1996.

[AG93] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, SE-19(1):24–40, January 1993.

[BC95] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with Binary Moment Diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 535–541, June 1995.

[BCG88] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 428–439. IEEE Computer Society Press, June 1990.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(6):677–691, August 1986.

[CDV96] J. Crow and B. L. Di Vito. Formalizing space shuttle software requirements. In *Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Practice*, pages 40–48, January 1996.

[CFZ95] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid Decision Diagrams overcoming the limitations of MTBDDs and BMDs. In *1995 IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers*, pages 159–163. IEEE Computer Society Press, November 1995.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HH95] T. A. Henzinger and P.-H. Ho. Algorithmic analysis of nonlinear hybrid systems. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 225–238. Springer-Verlag, July 1995.

[LHHR94] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, SE-20(9), September 1994.

[LS81] R. J. Lipton and R. Sedgewick. Lower bounds for VLSI. In *Conference Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, pages 300–307, May 1981.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[PB94] G. Pesant and M. Boyer. QUAD-CLP(ℝ): Adding the power of quadratic constraints. In *Second International Workshop, Principles and Practice of Constraint Programming*, pages 95–108. Springer-Verlag, May 1994.

[Tha96] J. S. Thathachar. On the limitations of ordered representations of functions. Technical Report CSE-96-09-03, University of Washington, September 1996.

[WME93] F. Wang, A. Mok, and E. A. Emerson. Symbolic model checking for distributed real-time systems. In *Proceedings of the First International Symposium of Formal Methods Europe*, pages 632–651, April 1993.