Automatic Verification of Hybrid Systems with Large Discrete State Space [*]

Werner Damm[1,2], Stefan Disch[3], Hardi Hungar[2], Jun Pang[1], Florian Pigorsch[3], Christoph Scholl[3], Uwe Waldmann[4], Boris Wirtz[1]

[1] Carl von Ossietzky Universität Oldenburg
Ammerländer Heerstraße 114-118, 26111 Oldenburg, Germany
[2] OFFIS e.V., Escherweg 2, 26121 Oldenburg, Germany
[3] Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 51, 79110 Freiburg, Germany
[4] Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany

Abstract. We address the problem of model checking hybrid systems which exhibit nontrivial discrete behavior and thus cannot be treated by considering the discrete states one by one, as most currently available verification tools do. Our procedure relies on a deep integration of several techniques and tools. A first-order extension of AND-Inverter-Graphs (AIGs) serves as a compact representation format for sets of configurations which are composed of continuous regions and discrete states. Boolean reasoning on the AIGs is complemented by first-order reasoning in various forms and on various levels. These include subsumption checks for simple constraints, test vector generation for fast inequality checks of boolean combinations of constraints, and an exact implication check. These techniques are integrated within a model checker for universal CTL. Technically, it deals with discrete-time hybrid systems with linear differentials. The paper presents the approach, the architecture of a prototype implementation, and first experimental data.

1 Introduction

The analysis of hybrid systems faces the difficulty of having to address not only the continuous dynamics of mechanical, electrical and other physical phenomena, but also the intricacies of discrete switching. Both of these two constituents of hybrid systems alone often pose a major challenge for verification approaches, and their combination is of course by no means simpler. For instance, the behavior of a car or airplane is usually beyond the scope of mathematically precise assessment, even if attention is restricted to only one particular aspect like the functioning of a braking assistant. Even though the continuous behavior might

[*] This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), `http://www.avacs.org/`

in such a case be rather simple – at least after it has been simplified by introducing worst-case assumptions to focus on the safety-critical aspects –, through the interaction with discrete state control the result is in most cases unmanageable by present-day techniques.

In this work, we address the analysis of hybrid systems with a focus on the discrete part. Systems with non-trivial discrete state spaces arise naturally in application classes where the overall control of system dynamics rests with a finite-state supervisory control, and states represent knowledge about the global system status. Examples of such global information encoded in states are phases of a cooperation protocol in inter-vehicle communication (such as in platooning maneuvers or in collision avoidance protocols), knowledge about global system states (e. g., on-ground, initial ascent, ascent, cruising, . . .), and/or information about the degree of system degradation (e. g., due to occurrence of failures). Jointly, these will govern the selection of appropriate maneuvers, carried out by a low-level "reflex layer". This reflex layer is the part of the control directly connected to the controlled plant. We call the states of this layer *modes* to distinguish them from the states of the supervisory control. While the selection of the modes depends on the supervisory control, there is no direct relation between the top-level discrete states and the continuous part.

In our approach, we intend to profit from the independence of the supervisory control states and the continuous sections. We attempt to do so by representing discrete states *symbolically*, as in symbolic model checking [5], and combine this with a first-order logic representation of the continuous part. In that way, unnecessary distinctions between discrete states can be avoided and efficiency gained.

This idea, which has already been pursued in a different setting in [14, 3], can be seen as combining symbolic model checking with Hoare's program logic [13]. The discrete part of the state is encoded in bit vectors of fixed length. Sets of discrete states are represented in an efficient format for boolean functions, in our case functionally reduced AND-inverter graphs (FRAIGs) [15]. The state vectors are extended by additional components referring to first-order conditions. Model checking works essentially as in [5, 17] on the discrete part, while in parallel for the continuous part a Hoare-like calculus is applied. An important detail is that the set of data conditions is dynamic: computing the effect of a system step usually entails the creation of new conditions.

To make an automatic proof procedure out of this, we add diverse reasoning procedures for the first-order components. Of central importance is the ability to perform a subsumption check on our hybrid state-set representation in order to detect whether a fixpoint has been reached during model checking. HySAT [9] is one of the tools we use for that purpose.

In its current form, our approach is applicable to checking universal temporal logic in discrete-time hybrid systems, where conditions and transitions contain linear terms over the continuous variables. This corresponds to a discretization of systems whose evolution is governed by linear differential equations, of which the linear hybrid automata from [11] form a subset.

We present our class of models formally in Section 2. Section 3 explains our procedure on a semantical and logical level. The implementation is described in Section 4, followed by a report on first experiments with our current prototype in Section 5. Sections 6 and 7 discuss related work and possible future extensions.

2 System Model

2.1 Discretization

As mathematical model we use discrete-time hybrid automata, which in each time step of fixed duration update a set of real-valued variables as determined by assignments occurring as transition labels. Since assignments and transition guards may use linear arithmetical expressions, this subsumes the capability to describe the evolvement of plant variables by difference equations. Steps of the automata are assumed to take a fixed time period (also called cycle-time), intuitively corresponding to the sampling period of the control unit, and determine the new mode and new outputs (corresponding to actuators) based on the sampled inputs (sensors).

The decision to base our analysis on discrete-time models of hybrid systems is motivated from an application perspective. Industrial design flows for embedded control software typically entail a transition from continuous time models in early analysis addressing control law design, to discrete-time models in modeling tools such as Scade$^{\text{TM}}$, ASCET$^{\text{TM}}$, or MATLAB/Simulink-StateFlow$^{\text{TM}}$, as a basis for subsequent autocode generation.

In this paper, we analyze closed loop systems with only discrete inputs, e. g., corresponding to discrete set points.

2.2 Formal model

Our analysis is based on discrete-time models of hybrid systems. Time is modeled implicitly, in that each step corresponds to a fixed unit delay δ, as motivated in the previous section.

We assume that a hybrid system operates over two disjoint finite sets of variables D and C. The elements of $D = \{d_1, \ldots, d_n, d_{n+1}, \ldots, d_\ell\}$ $(n \leq \ell)$ are discrete variables, which are interpreted over finite domains; $D_{in} = \{d_{n+1}, \ldots, d_\ell\} \subseteq D$ is a finite set of discrete inputs. The elements of $C = \{c_1, \ldots, c_m\}$ are continuous variables, which are interpreted over the reals $\mathbb{R}$. Let $\mathbf{D}$ denote the set of all valuations of D over the respective domains, $\mathbf{C} = \mathbb{R}^m$ the set of all valuations of C. The state space of a hybrid system is presented by the set $\mathbf{D} \times \mathbf{C}$; a valuation $(\mathbf{d}, \mathbf{c}) \in \mathbf{D} \times \mathbf{C}$ is a state of the hybrid system.

A set of states of a hybrid system can be represented symbolically using a suitable (quantifier-free) first-order logic formula over D and C. We assume that the data structure for the discrete variables D is given by a signature $\mathcal{S}_D$ which introduces typed symbols for constants and functions, and by $\mathcal{I}_D$ which assigns a meaning to symbols. We denote by $\mathcal{T}_D(D)$ the set of terms over

D, and by $\mathcal{B}(D)$ the set of boolean expressions over D. The first-order part on continuous variables is restricted to linear arithmetic of $\mathbb{R}$, which has the signature $\{\mathbb{Q}, +, -, \times, =, <, \leq\}$, where $\mathbb{Q}$ is the set of rational numbers appearing as constants, $\{+, -, \times\}$ is the set of function symbols, and $\{=, <, \leq\}$ is the set of predicate symbols. The interpretation $\mathcal{I}_C$ assigns meanings to these symbols as usual. We define $\mathcal{T}_C(C)$ as the set of linear terms over C. The set of first-order predicates $\mathcal{P}(C)$ over C can be defined according to the following syntax:

1. Atomic formulas: $\phi ::= p \sim 0$, where $\sim \in \{=, <, \leq\}$ and $p \in \mathcal{T}_C(C)$;
2. Boolean combinations: $\phi ::= \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi$.

We use $\psi(D)$, $\phi(C)$, $g(D)$, and $t(C)$, possibly with subscripts, to denote boolean expressions in $\mathcal{B}(D)$, first-order predicates in $\mathcal{P}(C)$, terms in $\mathcal{T}_D(D)$, and terms in $\mathcal{T}_C(C)$, respectively; D and C may be omitted, if they are clear from the context. We use $\mathcal{I}_D \vDash \psi(\mathbf{d})$ and $\mathcal{I}_C \vDash \phi(\mathbf{c})$ to denote that ψ and ϕ are true under the valuations $\mathbf{d}$ and $\mathbf{c}$. Thus $\psi \wedge \phi$ represents the sets of states of a hybrid system such that $\{ (\mathbf{d}, \mathbf{c}) \mid \mathcal{I}_D \vDash \psi(\mathbf{d}), \mathcal{I}_C \vDash \phi(\mathbf{c}) \}$. Assignments to the variables D and C are given in the form of $(d_1, \ldots, d_n) := (g_1, \ldots, g_n)$ and $(c_1, \ldots, c_m) := (t_1, \ldots, t_m)$; they may leave some variables unchanged.

Definition 1. A discrete-time hybrid system *DTHS contains four components:*

- $D = \{d_1, \ldots, d_n, d_{n+1}, \ldots, d_\ell\}$ *($n \leq \ell$) is a finite set of discrete variables,* $D_{in} = \{d_{n+1}, \ldots, d_\ell\} \subseteq D$ *is a finite set of discrete inputs;*
- $C = \{c_1, \ldots, c_m\}$ *is a finite set of continuous variables;*
- *Init is a set of initial states, given in the form of* $\psi_0 \wedge \phi_0$*, where* $\psi_0 \in \mathcal{B}(D - D_{in})$*;*
- *Trans is a union of a finite number of guarded assignments, each guarded assignment* ga_i *($i = 1, \ldots, k$ and $k \geq 1$) is in the form of*

$$\psi_i \wedge \phi_i \rightarrow (d_1, \ldots, d_n) := (g_{i,1}, \ldots, g_{i,n}); (c_1, \ldots, c_m) := (t_{i,1}, \ldots, t_{i,m}).$$

Each ga_i *is a deterministic transition, namely for every state* $(\mathbf{d}, \mathbf{c})$ *there is at most one state* $(\mathbf{d}', \mathbf{c}')$ *such that* $((\mathbf{d}, \mathbf{c}), (\mathbf{d}', \mathbf{c}')) \in ga_i$*. The assignment of* ga_i *transforms* $(\mathbf{d}, \mathbf{c})$ *to* $(\mathbf{d}', \mathbf{c}')$*. Moreover, such* $(\mathbf{d}', \mathbf{c}')$ *exists if and only if* $\mathcal{I}_D \vDash \psi_i(\mathbf{d})$ *and* $\mathcal{I}_C \vDash \phi_i(\mathbf{c})$*.*

We assume that the guards of the assignments defining the transition relation are exclusive and exhaustive. Nondetermism from the transition relation can be eliminated by introducing discrete *resolution* variables $R \subseteq D$.

A trajectory of a DTHS is a discrete-time sequence $(\mathbf{d}_i, \mathbf{c}_i)$ satisfying the conditions (i) $(\mathbf{d}_0, \mathbf{c}_0) \in \mathit{Init}$ and (ii) $((\mathbf{d}_i, \mathbf{c}_i), (\mathbf{d}_{i+1}, \mathbf{c}_{i+1})) \in \mathit{Trans}$ for all $i \in \{0, 1, \ldots\}$. Given a DTHS, we define the reachable set of states to be the set of all states that are reachable by a trajectory of the DTHS. The purpose of verification is to determine whether all possible behaviors of a system satisfy some property, which is specified as formula in a temporal logic.

3 Approach

3.1 Specification logic

We sketch a model checker for a temporal logic over discrete and quantifier-free first-order atoms. Though we could build, from our basic ingredients, a procedure handling full CTL (or a linear-time logic), we restrict ourselves to its universal fragment ACTL with the temporal operators $\mathbf{AX}\,\cdot$ (next), $\mathbf{A}[\cdot\,\mathbf{U}\,\cdot]$ (until) and $\mathbf{A}[\cdot\,\mathbf{W}\,\cdot]$ (unless), with $\mathbf{AG}\,\cdot$ (globally) and $\mathbf{AF}\,\cdot$ (finally) as derived operators.

In practice, we expect the valuations of continuous variables to come from bounded subsets of $\mathbb{R}$. In other words, for each $c \in C$ we assume a lower and an upper bound l_c and u_c. Such restrictions can be captured in *global constraints* GC. With global constraints present, the formula operators are interpreted as follows:

$$\begin{aligned}\mathbf{A}_{GC}\mathbf{X}\,\phi &= \neg GC \vee \mathbf{AX}\,(\phi \vee \neg GC)\\ \mathbf{A}_{GC}[\phi\,\mathbf{W}\,\psi] &= \mathbf{A}[\phi\,\mathbf{W}\,(\psi \vee \neg GC)]\\ \mathbf{A}_{GC}[\phi\,\mathbf{U}\,\psi] &= \mathbf{A}[\phi\,\mathbf{U}\,(\psi \vee \neg GC)]\end{aligned}$$

3.2 Representation by First-Order AND-Inverter Graphs

First-Order AND-Inverter Graphs (FO-AIGs) are the data structure that we use to represent predicates over states of hybrid systems in the model-checking procedure. States of hybrid systems consist of valuations of continuous variables C and discrete variables D. For ease of exposition we assume that discrete variables are encoded by sets of boolean variables, thus we consider in the following only continuous variables and boolean variables. Then a predicate over states of hybrid systems may be expressed by a set of first-order conditions and a boolean formula with two types of boolean variables: Boolean variables of the first type represent encodings of discrete variables; boolean variables of the second type are assigned to first-order conditions. We obtain the predicate described by this representation by replacing the variables of the second type by their corresponding first-order conditions.

In FO-AIGs boolean formulas are represented by Functionally Reduced AND-Inverter Graphs (FRAIGs) [15]. FRAIGs are basically boolean circuits consisting only of AND gates and inverters. In contrast to BDDs [3], they are not a canonical representation for boolean functions, but they are “semi-canonical” in the sense that every node in the FRAIG represents a unique boolean function. To achieve this goal several techniques like structural hashing, simulation and SAT solving are used [15, 17]. For the pure boolean case, enhanced with other techniques such as quantifier scheduling, node selection heuristics and BDD sweeping, FRAIGs proved to be a promising alternative to BDDs in the context of CTL model checking, avoiding in many cases the well-known memory explosion problem which may occur during BDD-based symbolic model checking [17].

The second component of FO-AIGs is the set of first-order conditions. In our implementation these conditions are restricted to *linear constraints on real variables.* We use normalization rules in order to identify equivalent conditions.

3.3 Step computation

Our procedure works backwards, which means that it has to compute pre-images of state sets. Since we are going to check ACTL, we compute

$$pre(S) =_{\mathrm{df}} \{\, s \mid \forall s'.\, s \rightarrow s' \Rightarrow s' \in S \,\}$$

which corresponds to the temporal operator **AX**. In general, computing the effect of a step would involve, on the data part, substitution and quantification (universal for **AX**), see [3]. Since we restrict ourselves to closed-loop systems, there are no continuous inputs. Thus we do not have to perform first-order quantification in a step: The effect on the data part is captured by a substitution. There are, however, discrete inputs to be treated via quantification at the end of the step computation.

Remember that the update of some $d_j \in D - D_{in}$ has the form of the following guarded choice.

$$[]_{i=1}^{k}\, \psi_i(D) \wedge \phi_i(C) \;\rightarrow\; d_j := g_{i,j}(D)$$

This translates to the update function:

$$pre(d_j) \;=\; \bigwedge_{i=1}^{k} (\psi_i(D) \wedge \phi_i(C) \;\rightarrow\; g_{i,j}(D))$$

For the continuous part, we do not have to update the variables C. Instead, the operation has to be performed on the set of first-order conditions. The transitions

$$[]_{i=1}^{k}\, \psi_i(D) \wedge \phi_i(C) \;\rightarrow\; (c_1, \ldots, c_m) := (t_{i,1}(C), \ldots, t_{i,m}(C))$$

induce

$$pre(q) \;=\; \bigwedge_{i=1}^{k} (\psi_i(D) \wedge \phi_i(C) \;\rightarrow\; q[c_1, \ldots, c_m / t_{i,1}(C), \ldots t_{i,m}(C)])$$

as an update for each condition q.

Finally, the pre-image of a set of states S is computed by substituting in parallel the pre-images for the respective variables, and afterwards universally quantifying over the discrete inputs.

$$pre(S) = \forall D_{in}.\, S[d_1, \ldots, d_n, q_1, \ldots, q_\ell \,/\, pre(d_1), \ldots, pre(d_n), pre(q_1), \ldots, pre(q_\ell)]$$

Note that the pre-image of a boolean variable is described by a quantifier-free formula which does not change during model checking – it can be computed once and for all. The same holds for each single condition variable: The right-hand side remains constant. But the RHS may contain conditions not already present in the current set of conditions represented as FRAIG variables. This necessitates to add condition variables during model checking, and also to add corresponding components to the step function. Intuitively, a condition might be a hyperplane serving as a bound to define a polyhedron in the continuous state space. The pre-image of the polyhedron then is bounded by other hyperplanes, whose descriptions are derived via substitution from the existing bounding conditions.

When computing a fixpoint approximatively, successively better approximations of the fixpoint are computed. In explicit or symbolic model checking, the criterion for detecting whether a fixpoint has been reached, is simple: Two successive approximations must be the same. Here, where constraints enter the state set descriptions, one has to check for *semantical* equality. This means that one has to check whether one boolean combination of conditions implies another (usually, one direction of the equivalence holds by construction). In our implementation, HySAT is used for this check. HySAT [9] is a tool for bounded model checking linear hybrid systems, which combines Davis-Putnam style SAT solving techniques with linear programming, using state of the art optimizations.

Note the procedure described above can be applied to a broad class of systems. The logical treatment of the step function permits arbitrary *linear* terms on the right-hand sides of assignments, like $c := \alpha_1 c + \alpha_2 c' + \alpha_0$. The linear hybrid automata from [11] can be discretized using only the more restricted format $c := c + \alpha$.

4 Realization

In order to implement the approach described in the previous section we use FO-AIGs for representing sets of states (as introduced in Sect. 3.2). Using efficient methods for keeping this representation as compact as possible is a key point for our approach. This goal is achieved by a rather complex interaction of various methods. In the following we give some more details on these concepts. The methods are divided into three classes:

- methods dealing with the boolean part,
- methods dealing with the first-order part, and
- methods dealing with the interaction of the boolean and the first-order part.

Note that algorithms for model checking ACTL formulas can be implemented in a straightforward manner, we omit the implementation details here.

4.1 Methods dealing with the boolean part

For the boolean part, we use a package implementing FRAIGs (Functionally Reduced AND-Inverter-Graphs) [17], which offers various mechanisms to keep the boolean part of the representation as compact as possible:

First, simple local transformation rules are used for node minimization. For instance, we apply structural hashing for identifying isomorphic AND nodes which have the same pairs of inputs.

Moreover, we maintain the so-called "functional reduction property": Each node in the FRAIG represents a unique boolean function (up to complementation). We use a SAT solver to check for equivalent nodes while constructing a FRAIG and to merge equivalent nodes immediately. Of course, checking each possible pair of nodes would be quite inefficient. However, *simulation* using test vectors restricts the number of candidates for SAT check to a great extent: If for a given pair of nodes simulation is already able to prove non-equivalence, more time consuming SAT checks are not needed. The simulation vectors are initially random, but they are updated using feedback from satisfied SAT instances (i. e., from proofs of non-equivalence).

4.2 Methods dealing with the first-order part

The second component of FO-AIGs is a representation of first-order conditions connected to the boolean part by boolean variables. As already mentioned in the previous section we restrict the first-order conditions to linear real arithmetic, i. e., our first-order conditions are linear constraints of the form $\sum_{i=1}^{n} \alpha_i c_i + \alpha_0 \sim 0$ with real constants α_j, real variables c_i, and $\sim \in \{=, <, \leq\}$. When new linear constraints are computed by substitution during the step computation (see Sect. 3), we avoid introducing new linear constraints which are equivalent to existing constraints. The restriction to *linear* constraints makes this task simple, since it reduces to the application of normalization rules.

4.3 Methods dealing with the interaction of the boolean and the first-order part

Of course, a strict separation between the boolean part and the first-order part of FO-AIGs gives us usually not enough information, for instance when we have to check whether two sets of states are equivalent during the fixpoint check of the model checking procedure. As a simple example consider the two predicates $P_1 = (c < 5)$ and $P_2 = (c < 10) \wedge (c < 5)$. If $c < 5$ is represented by the boolean variable a and $c < 10$ by variable b, then the corresponding boolean formulas a and $a \wedge b$ are not equivalent, whereas P_1 and P_2 are certainly equivalent. Both as a means for further compaction of our representations and as a means for detecting fixpoints we need methods for transferring knowledge from the first-order part to the boolean part. (In the example above this may be the information that $a = 1$ and $b = 0$ can not be true at the same time or that P_1 and P_2 are equivalent when replacing boolean variables by their first-order interpretations.)

Computing implications between linear constraints. In our first method we consider dependencies between linear constraints that are easy to detect a priori and transfer them to the boolean part. It is not known initially, which dependencies are actually needed in the rest of the computation; for this reason we restrict to two simple cases: First, we compute unconditional implications between linear constraints $\alpha_1 c_1 + \ldots + \alpha_n c_n + \alpha_0 \leq 0$ and $\alpha_1 c_1 + \ldots + \alpha_n c_n + \alpha_0' \leq 0$, where $\alpha_0 > \alpha_0'$ (and analogously implications involving negations of linear constraints). Second, we consider implications modulo global constraints, where a linear constraint $\alpha_1' c_1 + \ldots + \alpha_n' c_n + \alpha_0' \leq 0$ follows from $\alpha_1 c_1 + \ldots + \alpha_n c_n + \alpha_0 \leq 0$ and the global lower and upper bounds $l_i \leq c_i \leq u_i$ for the first-order variables.

Using implications between linear constraints. Suppose we have found a pair of linear constraints lc_1 and lc_2 with $lc_1 \rightarrow lc_2$, and in the boolean part lc_1 is represented by variable a, lc_2 by variable b. Then we know that the combination of values $a = 1$ and $b = 0$ is inconsistent w. r. t. the first-order part, i. e., it will never be applied to inputs a and b of the boolean part. We transfer this knowledge to the boolean part by a modified behavior of the FRAIG package: First we adjust our simulation vectors, replacing vectors with $a = 1$ and $b = 0$ by corresponding vectors with $a = 1$ and $b = 1$ (potentially leading to the fact that proofs of non-equivalence by simulation will not hold any longer for certain pairs of nodes) and second we introduce the implication $a \rightarrow b$ as an additional clause in every SAT problem checking equivalence of two nodes depending on a and b. In that way non-equivalences of AIG nodes which are only caused by differences w. r. t. inconsistent input value combinations with $a = 1$ and $b = 0$ will be turned into equivalences, removing redundant nodes in the AIG.

Using a decision procedure for deciding equivalence. In addition to the eager dependency check for linear constraints, we use HySAT [9] as a decision procedure for the equivalence of nodes in FO-AIGs (representing boolean combinations of linear constraints). If two nodes are proven to be equivalent (taking the linear constraints into account), then these nodes can be merged, leading to a compaction of the representation or leading to the detection of a fixpoint in the model checking computation.

In principle, we could use HySAT in an eager manner every time when a new node is inserted into the FO-AIG representation, just like SAT (together with simulation) is used in the FRAIG representation of the boolean part. This would lead to a FO-AIG representation where different nodes in the FRAIG part always represent different first-order predicates. However, we decided to use HySAT only in a lazy manner (i. e., only from time to time) in order to avoid too many potentially expensive applications of HySAT (taking the linear constraints into account). The HySAT checks used in our first implementation include the fixpoint checks of the model checking procedure.

Using test vectors to increase efficiency. As in the boolean case (see Sect. 4.1), we use simulation with test vectors as an incomplete but cheap method to show the non-equivalence of FO-AIG nodes, thus reducing the number of expensive calls to HySAT. However, note that the boolean simulation vectors which we apply to the boolean variables corresponding to linear constraints must now be *consistent with respect to the linear constraints*, since otherwise our proof of non-equivalence could be incorrect. This means that we need an appropriate set of test vectors in terms of real variables leading to consistent boolean simulation vectors.

Trying to find an optimal set of test vectors that allows us to distinguish between any two boolean combination of linear constraints is at least as hard as solving our main problem, the implication check between such boolean combinations, and therefore unpractical. On the other hand, if test vectors are picked randomly with a uniform distribution over the polyhedron of permitted values, a large number of them fall into “uninteresting regions” of this polyhedron.

Our solution is to choose test vectors randomly *in the proximity of relevant hyperplanes:* Assume that every variable c_i has a global lower and upper bound $l_i \le c_i \le u_i$, so that the polyhedron of permitted values is $Q = \{\vec{c} \mid \vec{c} = (c_1, \dots, c_n),\, l_i \le c_i \le u_i\}$. For each linear constraint $f(\vec{c}) \le 0$ with $f(\vec{c}) = \alpha_1 c_1 + \dots + \alpha_n c_n + \alpha_0$ we determine a set of test vectors by first computing random points $\vec{t} \in Q$ and then interpolating between $\vec{t}$ and $\vec{r}$ (if $f(\vec{t}) < 0$) or $\vec{t}$ and $\vec{s}$ (otherwise), where $\vec{r}$ and $\vec{s}$ are the vertices of Q for which f is maximal or minimal, respectively. (Without loss of generality, $f(\vec{r}) > 0 > f(\vec{s})$.)

Similarly to the boolean case, we learn additional boolean simulation vectors from satisfied HySAT instances (i. e., from proofs of non-equivalence). The satisfying assignments computed by HySAT are guaranteed to be consistent w. r. t. linear constraints and they are able to separate at least the pair of nodes which are currently proven to be non-equivalent.

4.4 Software architecture

Figure 1 gives an overview of the software architecture of our implementation and summarizes the various components described in the last two sections. Arrows depict the flow of information between the different parts.

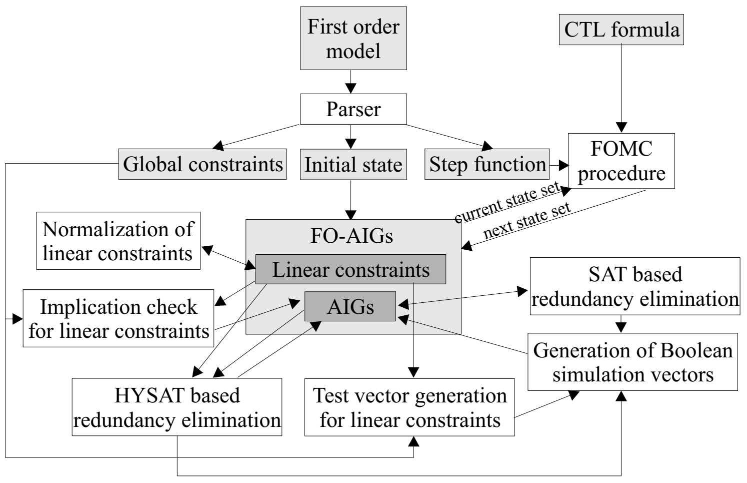
First order model
CTL formula
Parser
Global constraints
Initial state
Step function
FOMC procedure
current state set
next state set
Normalization of linear constraints
FO-AIGs
Linear constraints
AIGs
Implication check for linear constraints
SAT based redundancy elimination
Generation of Boolean simulation vectors
HYSAT based redundancy elimination
Test vector generation for linear constraints

Fig. 1. Software architecture. Gray shaded boxes represent data structures, white boxes represent algorithmic methods.

5 Experimental Results

We implemented a prototype model checker based on the concepts mentioned above and applied it both to several small examples taken from literature and to a model derived from an industrial case study. In this section we will report on results for the case study; more details about this example (a controller for the flaps of an aircraft [4]) are given in Appendix A. We were able to successfully model check the flap controller showing that the system remains in the safe region. Using all the concepts presented in Section 4 our model checking run was completed after 46 steps within 4.7 minutes of CPU time.[5]

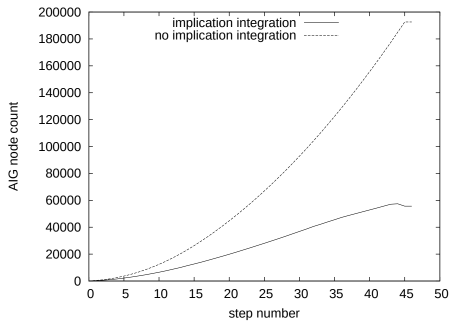

Fig. 2. Number of AIG nodes.

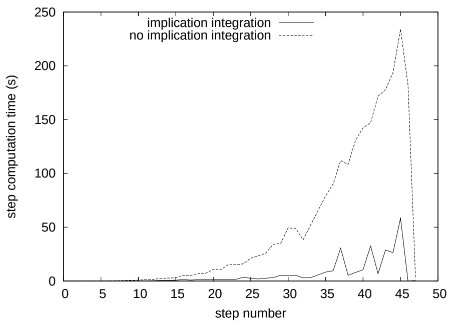

Fig. 3. CPU times for different steps.

In a first experiment we evaluated the effect of integrating knowledge of implications between linear constraints into the FO-AIG representation. We compared two cases: Case *no_impl* when no implications were computed and integrated and case *impl* when implications were computed and integrated as described in Section 4.3. Figure 2 depicts the number of AIG nodes used during the different steps of the model checking procedure both for case *no_impl* (dashed line) and case *impl* (solid line). For case *no_impl* the maximal number of active AIG nodes was $192,630$ whereas for case *impl* the maximal number was only $59,372$. This clearly shows that integrating knowledge of linear constraints pays off in terms of node counts: By using implications it was possible to simplify the representation to a great extent, since AIG nodes were identified which were equivalent taking the linear constraints into account. Figure 3 shows that making use of implications not only improves node counts, but run times as well: It presents the run times needed for the different steps in both cases. The total run time for case *no_impl* was 37.9 CPU minutes whereas the total run time for case *impl* was 4.7 CPU minutes. (The total number of implications between linear constraints computed by our tool was 622 (implications due to transitivity not taken into account).)

[5] All experiments were performed on a dual Opteron 250, 2.4 GHz with 4 GB memory.

In the following we will confine ourselves to case *impl* and we will perform a more detailed analysis of the behavior of our representation of states containing discrete and continuous variables. The efficiency of our FO-AIGs relies both on efficient methods for Boolean manipulations and on efficient methods for integrating knowledge of linear constraints avoiding the application of more expensive calls to a linear constraint solver as much as possible.

We could observe that the number of SAT checks divided by the total number of attempts to insert a node into the FRAIG was only 0.24 % in our experiment. The fraction of SAT checks which led to the result that the compared nodes were functionally equivalent was 59 %. This means that – although we are always maintaining the functional reduction property of FRAIGs – the assistance of SAT by simulation and structural hashing as described in Sect. 4.1 assures that SAT is applied only for a small fraction of all node insertions. Moreover, the high percentage of SAT checks proving functional equivalence of two nodes shows the effectiveness of simulation in avoiding unnecessary SAT checks for nodes which are not equivalent.

In a last experiment we analyzed how often the application of calls to the linear constraint solver HySAT was saved by incomplete (but inexpensive) methods. In our method HySAT calls can be saved for two reasons:

1. The equivalence of two Boolean combinations of linear constraints can be proven just by considering the Boolean part (without interpreting the variables representing linear constraints).
2. The non-equivalence of two Boolean combinations of linear constraints can be proven by simulation with test vectors as described in Section 4.3.

Our model checking run involved 5374 equivalence checks for Boolean combinations of linear constraints. However, for only 22 out of these 5374 checks it turned out to be necessary to call the linear constraint solver in HySAT (i. e., in 0.41 % of all cases). In 42.91 % of all cases the call to HySAT could be avoided due to reason (1) and in 56.68 % of all cases due to reason (2).

Although we believe that the complex interaction of different methods in our approach to first-order model checking still leaves room for improvement, our first experiments provide promising results confirming our idea of increasing efficiency by incomplete but inexpensive methods.

6 Related Work

We address hybrid systems consisting not only of a continuous part, but also of a potentially complex discrete part. Tools like HyTech [12], `d/dt` [1], PHAver [10] based on the notion of hybrid automaton [11] fail when dealing with complex hybrid controllers, since only the continuous part of the system is represented symbolically, while the discrete states are represented explicitly. Thus, these tools cannot take advantage of the breakthrough achieved for symbolic model checkers [5]. In this section, we discuss those verification tools which can (potentially) deal with hybrid systems with large discrete parts, and compare them with our work in the end of this section.

CheckMate [19] is a MATLAB-based tool for simulation and verification of *threshold-event driven hybrid systems* (TEDHSs). A TEDHS has a clear separation between purely continuous blocks representing the dynamics in a given mode and discrete controllers. The changes in the discrete state can occur only when continuous state variables encounter specified thresholds. CheckMate converts the TEDHS model into a *polyhedral-invariant hybrid automaton* [6], computes the sets of reachable states for the continuous dynamics using *flowpipe* approximations [7], and performs search in a completely constructed *approximate quotient transition system.* This approach was adapted for discrete-time controllers with fixed sampling rate [18], where the sampled behavior only applies to conditions for discrete-state transitions.

Separation of continuous dynamics and control by observing threshold predicates as guards of transitions was also taken in [3, 2], which extended symbolic model checking with dynamically generated first-order predicates. Those predicates express sets of valuations over large data domains like reals. BDDs are used to encode discrete states, and specific variables within the BDDs are used to present those first-order formulas, which are maintained separately.

The SAL verification tool [16] for hybrid systems builds on a symbolic representation of polynomial hybrid systems in PVS, the guards on discrete transitions and the continuous flows in all modes can be specified using arbitrary polynomial expressions over the continuous variables. SAL applies *hybrid abstraction* [20] to construct a sound discrete approximation using a set of polynomial expressions to partition the continuous state space into *sign-invariant zones.* This abstract discrete system is passed to a symbolic model checker. SAL also uses other techniques like quantifier elimination and invariant generation.

HySAT [9] is a bounded model checker for linear hybrid systems. It combines Davis-Putnam style SAT solving techniques with linear programming, and implements state of the art optimizations such at nonchronological backjumping conflict driven learning and lazy clause evaluation.

HYSDEL [21] is a model language for describing discrete-time hybrid systems by interconnections of linear dynamic systems, finite-state automata, if-then-else and propositional logic rules. The description can be transformed into a Mixed Logical Dynamical (MLD) system. HYSDEL uses mathematical programming to perform reachability analysis for MLD systems. The algorithms determines the reachable set by solving a mixed-integer optimization problems.

Both CheckMate and SAL construct a discrete approximation in order to perform model checking. Our approach checks properties directly on a computed reachable state space, which includes both discrete and continuous parts, without using any approximation. Moreover, instead of using BDDs as in [3, 2], we use FO-AIGs as symbolic representation of hybrid state spaces. Various techniques like implication test and test vector generation are tightly integrated to identify equivalent and non-equivalent linear constraints efficiently. This approach allows us to deal with large discrete state spaces, while smoothly incorporating reasoning about continuous variables (linear constraints). From this perspective, our approach is different with all the aforementioned works. Unlike bounded model checking in HySAT, we perform verification on a completely constructed state space. Tools like CheckMate and SAL deal with continuous-time hybrid systems. Our approach focuses on discrete-time hybrid systems as HYSDEL, but the analysis procedure in HYSDEL is different from ours.

7 Concluding Remarks

In this paper, we have proposed an approach for model checking safety propreties of discrete-time hybrid systems. It uses a first-order extension of AIGs as a compact representation for sets of configurations, which are composed of both continuous regions and discrete states. Several efficient methods for keeping this representation as compact as possible have been tightly integrated. For instance, we have implemented techniques to keep the discrete part functionally reduced, to detect implications between linear constraints, to use a decision procedure to perform equivalence checks on our hybrid state-set representation, to generate test vectors to distinguish between any two boolean combination of linear constraints. The typical application domain of our approach is hybrid systems with non-trivial discrete state spaces.

The implementation of our approach has only been used to check an industrial case study with limited size and several small examples taken from the literature. More sophisticated examples will be checked within our project. We expect to report more experience, and to compare a mature implementation with other tools (see Sect. 6). Our tool computes the set of reachable states exactly. Our previous works [3, 2] have already used predicate abstraction to derive a finite-state abstraction of the hybrid system either on-the-fly or at a separate initial abstraction step. These techniques can be integrated without much difficulties. The exectution time of our tool heavily relies on discretization. Currently, techniques like *acceleration* to speed up step computation are under our investigation. We also plan to use counter-example guided abstraction refinement, as it has been added to CheckMate [8] recently.

References

1. E. Asarin, T. Dang, and O. Maler. The d/dt tool for verification of the hybrid systems. In *14th Conference on Computer Aided Verification*, 2002, *LNCS 2404*, pp. 365–370. Springer.
2. T. Bienmüller, J. Bohn, H. Brinkmann, U. Brockmeyer, W. Damm, H. Hungar, and P. Jansen. Verification of the automotive control units. In *Correct System Design – Recent Insights and Advances*, 1999, *LNCS 1710*, pp. 319–341. Springer.
3. J. Bohn, W. Damm, O. Grumberg, H. Hungar, and K. Laster. First-order-CTL model checking. In *18th Conference on Foundations of the Software Technology and Theoretical Computer Science*, 1998, *LNCS 1530*, pp. 283–294. Springer.

4. M. Bretschneider, H.-J. Holberg, E. Böde, I. Brückner, T. Peikenkamp, and H. Spenke. Model-based safety analysis of a flap control system. In *14th Annual INCOSE Symposium*, 2004.
5. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, 1990, pp. 428–439. IEEE Press.
6. A. Chutinan and B. H. Krogh. Computing polyhedral approximations to flow pipes for dynamic systems. In *37th IEEE Conference on Decision and Control*, 1998. IEEE Press.
7. A. Chutinan and B. H. Krogh. Verification of the polyhedral-invariant hybrid automata using polygonal flowpipe approximations. In *2nd Workshop on Hybrid Systems: Computation and Control*, 1999, *LNCS 1569*, pp. 76–90. Springer.
8. E. M. Clarke, A. Fehnker, Z. Han, B. H. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Foundations of Computer Science*, 14(4), 2003.
9. M. Fränzle and C. Herde. Efficient proof engines for bounded model checking of hybrid systems. *ENTCS*, 133:119–137, 2005.
10. G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In *8th Workshop on Hybrid Systems: Computation and Control*, 2005, *LNCS 3414*, pp. 258–273. Springer.
11. T. A. Henzinger. The theory of hybrid automata. In *11th IEEE Symposium on Logic in Computer Science*, 1996, pp. 278–292. IEEE Press.
12. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1-2), 1997.
13. C.A.R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 12:576–583, 1969.
14. H. Hungar, O. Grumberg, and W. Damm. What if model checking must be truly symbolic. In *8th Conference on Correct Hardware Design and Verification Methods*, 1995, *LNCS 987*, pp. 1–20. Springer.
15. A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., UC Berkeley, 2005.
16. L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *16th Conference on Computer-Aided Verification*, 2004, *LNCS 3114*, pp. 496–500. Springer.
17. F. Pigorsch, C. Scholl, and S. Disch. Advanced unbounded model checking by using AIGs, BDD sweeping and quantifier scheduling. In *9th ITG/GI/GMM Workshop 'Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen'*, 2006.
18. B. I. Silva and B. H. Krogh. Modeling and verification of hybrid system with clocked and unclocked events. In *40th Conference on Decision and Control*, 2001.
19. B. I. Silva, K. Richeson, B. H. Krogh, and A. Chutinan. Modeling and verification of hybrid dynamical system using CheckMate. In *4th Conference on Automation of Mixed Processes*, 2000.
20. A. Tiwari and G. Khanna. Series of the abstractions for hybrid automata. In *5th Workshop on Hybrid Systems: Computation and Control*, 2002, *LNCS 2289*, pp. 465–478. Springer.
21. F. D. Torrisi and A. Bemporad. HYSDEL – A tool for generating computational hybrid models. *IEEE Transactions on Control Systems Technology*, 12(2):235–249, 2004.

A Appendix: Case study

In this appendix we give some more details concerning the example used in Section 5. The example is derived from an industrial case study, a controller for the flaps of an aircraft [4]. The flaps are extended during take-off and landing to generate more lift at low velocity. They are not robust enough for high velocity, so they must be retracted. It is the controller's task to correct the pilot's commands if he endangers the flaps. In the industrial case study performed for Airbus, a rather elaborate model was analyzed. We extracted a simplified system with four components to model, i. e., the pilot behavior, the controller, the flap mechanism, and the rest of the aircraft. It contains two continuous variables v (velocity) and f (flap angle), and two discrete variables ℓ (lever position set by the pilot) and c (corrected position, set by the controller). For each lever position, there is a pre-defined flap position and a pre-defined nominal (maximal) velocity.

A typical aircraft-level safety requirement related to the example is the following: "For the current flap setting, CAS shall not exceed VF", where CAS is the Calibrated Air Speed, modeled by v in our case, and VF is the maximum allowed velocity for a given flap position plus 7 knots.

The flap controller is not supposed to guarantee safety under all circumstances. The pilot gets corrected, but not eliminated. To enable manoeuvres risking aircraft integrity in critical situations, the controller is limited to only modify the pilot's command by one notch. Therefore, the property we aim to establish is that the flaps will always be in a safe position *provided the pilot acts reasonably*, that is, if he deviates only moderately from a safe behavior. Whether this requirement holds for our model depends on a "race" between flap retraction and speed increase. The controller is correct, if it initiates flap retraction (by correcting the pilot) early enough.

The pilot component in our model ensures "reasonable" lever positions, by guaranteeing that the lever is at most one notch too high. The behavior of the controller depends on both ℓ and v: When the velocity is greater than the nominal max value ($\mathit{nominal} + \mathit{slack}$), the modification of the pilot behavior is activated ($c = \ell - 1$); when the velocity has changed to less than the nominal min value ($\mathit{nominal} - \mathit{slack}$), the modification is turned off ($c = \ell$). The flap mechanism controls the continuous variable f, and depends on the discrete variable c. It models the mechatronic which adapts the physical flap angle f to the position commanded by c. This is a process which takes time. f has a range from 0 to 55.0. At each discrete time step (the sampling rate is $\delta = 100\,\text{ms}$ in this example), the flap angle may change by $\Delta_f = 0.15625$. At the same time, the rest of the aircraft might increase the velocity by 0.5 knots within a range from 150.0 to 340.0 knots.

This defines the "races" mentioned above. Our specification of the model is simply $\mathbf{AG}\,\mathit{safe}$. Our current prototype successfully model checked the flap controller with 3 lever positions, $220.0 \leq v \leq 340.0$, and $0.0 \leq f \leq 20.0$, showing that the system remains in the safe region.