Preprint from Proceedings of Fourth International Symposium on
Automated Technology for Verification and Analysis, October 2006, Beijing, China

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], and 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. An extension of AND-Inverter-Graphs (AIGs) with first-order constraints 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 implication checks for simple constraints, test vector generation for fast inequality checks of boolean combinations of constraints, and an exact subsumption check for representations of two configurations.

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, its 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/`).

S. Graf and W. Zhang (Eds.): ATVA 2006, LNCS 4218, pp. 276–291, 2006.
© Springer-Verlag Berlin Heidelberg 2006

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, . . . for an aircraft), and/or information about the degree of system degradation (e. g., due to occurrence of failures). States of the control determine the selection of appropriate continuous maneuvers, and conditions on the continuous state (reached thresholds, for instance) trigger changes in the control. But while there might be tens or hundreds of boolean state variables, often there are only very few different maneuvers and continuous trigger conditions, so that much of the discrete switching happens independently of the continuous evolution.

In our approach, we intend to profit from the independence of the supervisory control and the continuous sections, using adequate techniques for each of the two constituents in a hybrid procedure. We 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 linear (first-order) constraints. 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 constraints is dynamic: computing the effect of a system step usually entails the creation of new constraints. So it is not just model checking a finite-state encoding of the hybrid verification problem.

To make an automatic proof procedure out of this, we add diverse reasoning procedures for the first-order constraints. 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. However, a key point of our approach is the idea to avoid expensive applications of decision procedures as much as possible. Test vector generation for fast inequality checks of boolean combinations of constraints, implication checks for linear constraints, and advanced boolean reasoning are examples for methods which provide some lightweight and

inexpensive reasoning and are used both in the context of subsumption checks and for keeping state set representations as compact as possible.

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. These correspond to a time 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 Time 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 (from 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™, ASCET™, or Matlab/Simulink-StateFlow™, as a basis for subsequent autocode generation. We address the latter class of models, from which the production code can be generated. Note that the discrete complexity of our systems results mainly from the control logic, discretization of time only adds one more dimension to the complexity.

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 $D = \{d_1, \ldots, d_n, d_{n+1}, \ldots, d_p\}$ $(n \leq p)$ are discrete variables, which are interpreted over finite domains; $D_{in} = \{d_{n+1}, \ldots, d_p\} \subseteq D$ 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,
- $\mathcal{L}(C)$ as the set of linear constraints, with the syntax $t \sim 0$, where $\sim \in \{=, <, \leq\}$ and $t \in \mathcal{T}_C(C)$, and
- $\mathcal{P}(D, C)$, the set of first-order predicates, as boolean combinations of expressions in $\mathcal{B}(D)$ and linear constraints.

We use $\phi(D, C)$, $g(D)$, $t(C)$, and $\ell(C)$, possibly with subscripts, to denote first-order predicates in $\mathcal{P}(D, C)$, terms in $\mathcal{T}_D(D)$, terms in $\mathcal{T}_C(C)$, and linear constraints in $\mathcal{L}(C)$, respectively; D and C may be omitted, if they are clear from the context. We use $\mathcal{I}_{D,C} \models \phi(\mathbf{d}, \mathbf{c})$ to denote that ϕ is true under the valuations $\mathbf{d}$ and $\mathbf{c}$. Thus ϕ represents the sets of states of a hybrid system such that $\{ (\mathbf{d}, \mathbf{c}) \mid \mathcal{I}_{D,C} \models \phi(\mathbf{d}, \mathbf{c}) \}$. 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:*

- *$D = \{d_1, \ldots, d_n, d_{n+1}, \ldots, d_p\}$ ($n \leq p$) is a finite set of discrete variables, $D_{in} = \{d_{n+1}, \ldots, d_p\} \subseteq D$ 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 $\phi_0(D - D_{in}, C)$;*
- *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*

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

The assignment of ga_i transforms a state $(\mathbf{d}, \mathbf{c})$ to $(\mathbf{d}', \mathbf{c}')$. Moreover, such $(\mathbf{d}', \mathbf{c}')$ exists if and only if $\mathcal{I}_{D,C} \models \phi_i(\mathbf{d}, \mathbf{c})$.

We assume that the guards of the assignments defining the transition relation are exclusive and exhaustive. This is no restriction of the set of systems we consider, as nondetermism can be eliminated from the transition relation by introducing

resolution variables $R \subseteq D$. These are discrete inputs which are used like r in the following illustration of the case of two overlapping guards:

$$\left.\begin{matrix} \phi_1 \to \mathit{assignment}_1 \\ \phi_2 \to \mathit{assignment}_2 \end{matrix}\right\} \rightsquigarrow \left\{\begin{matrix} \phi_1 \wedge (\neg\phi_2 \vee r = 1) \to \mathit{assignment}_1 \\ \phi_2 \wedge (\neg\phi_1 \vee r = 2) \to \mathit{assignment}_2 \end{matrix}\right.$$

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 Logical Representation of State Sets

Our model-checking procedure operates on logical representations of state sets. For ease of exposition we assume that discrete variables are encoded by sets of boolean variables, i. e., we consider D as a set of boolean variables. Then, a state-set representation is a boolean formula over D and $\mathcal{L}(C)$, the set of linear constraints. To be able to use advanced data structures for boolean formulas, we introduce a set of new (boolean) *constraint variables* Q as encodings for linear constraints, where each occurring $\ell \in \mathcal{L}(C)$ is represented by some $q_\ell \in Q$. Thus we arrive at boolean formulas over $D \cup Q$, together with a mapping of Q into $\mathcal{L}(C)$.

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** ($\Rightarrow$ stands for logical implication, $\rightarrow$ for the transition relation). On the logical level, for the transitions of our DTHSs consisting of conditions, assignments and input, this can be expressed by substitution for assignments and universal quantification for input, see [3]. Since we have restricted ourselves to closed-loop systems, there are no continuous inputs. Therefore, there is no need for first-order quantification, only boolean quantification has to be performed. In the following, we describe in detail how to compute *pre* for our state-set representations, given a DTHS. The variables in D and Q are treated rather differently.

A discrete variable $d_j \in D - D_{in}$ is updated according to the transitions in the following set.

$$\{ \phi_i(D, C) \ \rightarrow \ d_j := g_{i,j}(D) \mid i = 1, \ldots k \}$$

This translates to the (logical) update function:

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

For the continuous part, we have to update the variables Q. The transitions

$$\{ \phi_i(D, C) \ \rightarrow \ (c_1, \ldots, c_m) := (t_{i,1}(C), \ldots, t_{i,m}(C)) \mid i = 1, \ldots k \}$$

induce

$$pre(q_\ell) \ = \ \bigwedge_{i=1}^{k} \left(\phi_i(D, C) \ \Rightarrow \ q_{\ell[c_1,\ldots,c_m/t_{i,1}(C),\ldots t_{i,m}(C)]}\right)$$

as an update for a constraint variable q_ℓ occurring in the state-set description. That is, each q_ℓ gets replaced by a boolean combinations of constraint variables. In this formula, $q_{\ell[c_1,\ldots,c_m/t_{i,1}(C),\ldots,t_{i,m}(C)]}$ is a (possibly new) constraint variable which represents the linear constraint resulting from ℓ by replacing the variables c_j by the terms $t_{i,j}(C)$.

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.

$$\begin{aligned} &pre(S) = \\ &\quad \forall D_{in}. S[d_1, \ldots, d_n, q_{\ell_1}, \ldots, q_{\ell_v} \, / \, pre(d_1), \ldots, pre(d_n), pre(q_{\ell_1}), \ldots, pre(q_{\ell_v})] \end{aligned}$$

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 constraint variable: The right-hand side remains constant. But the RHS may contain a constraint not already present in the

formula. This necessitates to add constraint variables to the state representations during model checking, and also to add corresponding components to the step function. This corresponds to the semantical view of model-checker steps: Semantically, an occurring constraint is 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.

3.4 Model Checking

The computation of the effect of a step is one main ingredient of CTL model checking. Besides that, one needs the ability to check whether two sets of states are equal, to detect that a fixpoint has been reached. In explicit or symbolic model checking, the criterion is simple: Two successive approximations must be the same. Here, where constraints enter the state-set descriptions, one has to check for *semantical* equality. Since our constraints are linear, this problem is decidable. This check for implication between two state-set representation completes the model-checking procedure.

In the following section we will present how we realized the conceptual procedure of this section, explaining the concrete representation format, how we perform logical operations and test for semantical implication.

Remark 2. 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 $c_1 := \alpha_1 c_1 + \alpha_2 c_2 + \alpha_0$. Discretization of the linear hybrid automata from [11] yields the more restricted format $c := c + \alpha$.

4 Realization

In order to implement the approach described in the previous section, we use a new data structure for representing sets of states, the so-called First-Order AND-Inverter-Graphs (FO-AIGs) (see Fig. 1 for an illustration).

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 to implement the model-checking algorithm we need only boolean operations, substitution and first-order implication. Our description focuses on the the first-order part and on how to keep our data structures small.

4.1 Methods Dealing with the Boolean Part

In FO-AIGs boolean formulas are represented by Functionally Reduced AND-Inverter Graphs (FRAIGs) [15,17]. FRAIGs are basically boolean circuits consisting only of AND gates and inverters. In contrast to BDDs as used in [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:

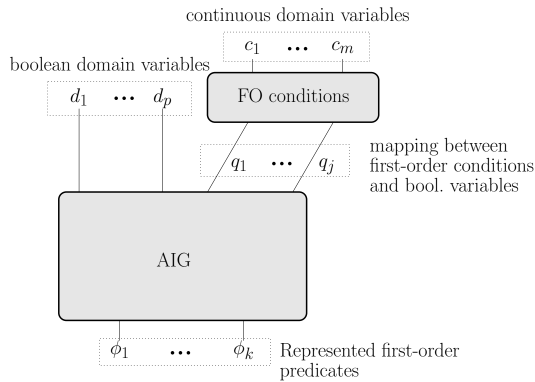

Fig. 1. The FO-AIG structure

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 of boolean values 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 (i. e., the simulated values are different for at least one test vector), the 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).

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].

4.2 Methods Dealing with the First-Order Part

The second component of FO-AIGs is a representation of linear constraints ℓ connected to the boolean part by constraint variables q_ℓ. These constraints are of the form $\sum_{i=1}^{n} \alpha_i c_i + \alpha_0 \sim 0$ with rational 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 linear constraints

which are equivalent to existing constraints. The restriction to *linear* constraints makes this task simple, since it reduces to the application of (straightforward) 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 $\phi_1 = (c < 5)$ and $\phi_2 = (c < 10) \wedge (c < 5)$. If $c < 5$ is represented by the boolean constraint variable q_1 and $c < 10$ by variable q_2, then the corresponding boolean formulas q_1 and $q_1 \wedge q_2$ are not equivalent, whereas ϕ_1 and ϕ_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 $q_1 = 1$ and $q_2 = 0$ can not be true at the same time or that ϕ_1 and ϕ_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 use a sound but incomplete method to detect 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 ℓ_1 and ℓ_2 with $\ell_1 \Rightarrow \ell_2$, and in the boolean part ℓ_1 is represented by the constraint variable q_1, ℓ_2 by variable q_2. Then we know that the combination of values $q_1 = 1$ and $q_2 = 0$ is inconsistent w. r. t. the first-order part, i. e., it will never be applied to inputs q_1 and q_2 of the boolean part. We transfer this knowledge to the boolean part by a modified behavior of the FRAIG package: First we adjust our test vectors, such that they become consistent with the found implications (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 $q_1 \Rightarrow q_2$ as an additional clause in every SAT problem checking equivalence of two nodes depending on q_1 and q_2. In that way non-equivalences of AIG nodes which are only caused by differences w. r. t. inconsistent input value combinations with $q_1 = 1$ and $q_2 = 0$ 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 above, 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 in order to avoid too many potentially expensive applications of HySAT (taking the linear constraints into account): In our first implementation HySAT is only invoked by explicit equivalence checks and 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. For this reason we use an appropriate set of test vectors in terms of real variables such that we can compute consistent boolean valuations of linear constraints based on the real valued test 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 $l_i \leq c_i \leq u_i$, so that the polyhedron of permitted values is $P = \{ \vec{c} \mid \vec{c} = (c_1, \ldots, c_n), l_i \leq c_i \leq u_i \}$. For each linear constraint $f(\vec{c}) \leq 0$ with $f(\vec{c}) = \alpha_1 c_1 + \ldots + \alpha_n c_n + \alpha_0$, we determine first the vertices $\vec{r}$ and $\vec{s}$ of P for which f is maximal or minimal, respectively (without loss of generality, $f(\vec{r}) > 0 > f(\vec{s})$). Second, we compute random points $\vec{t} \in P$, and finally, for each of these random points, we use linear interpolation between $\vec{t}$ and $\vec{r}$ (if $f(\vec{t}) < 0$) or $\vec{t}$ and $\vec{s}$ (otherwise) to obtain a point on the straight line between $\vec{t}$ and $\vec{r}$ (or $\vec{t}$ and $\vec{s}$) that is close to the hyperplane defined by $f(\vec{c}) = 0$.

Satisfied HySAT instances (i. e., proofs of non-equivalence for boolean combinations of linear constraints) are another source of boolean simulation vectors which are consistent w. r. t. linear constraints. 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. (Learning from HySAT corresponds to learning from SAT in the pure Boolean case.)

5 Application

We implemented a prototype model checker based on the concepts mentioned above and applied it both to several small examples and to a model derived from an industrial case study. In this section we report on results for the case study.

5.1 The Case Study

General Description. Our sample application is derived from a case study for Airbus, 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 for cruising period. It is the controller's task to correct the pilot's commands if he endangers the flaps. However, the flap controller is not supposed to guarantee safety under all circumstances, but only if the pilot acts "reasonably". To enable manoeuvres risking aircraft integrity in critical situations, the controller is limited to only modify the pilot's command by one notch.

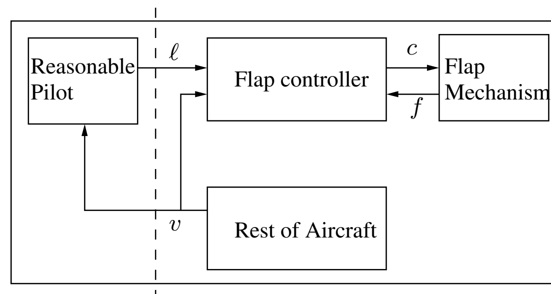

Fig. 2. Components in the flap controller example

Model Structure. Our simplified system consists of 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 velocity $nominal(f)$.

Property. The property "safe" to establish for our model is the following: "For the current flap setting f, the aircraft's velocity v shall not exceed the nominal velocity $nominal(f)$ plus 7 knots". 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.

Model Details. 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 ($nominal(f) + 2.5$ knots), the modification of the pilot behavior is activated ($c = \ell - 1$); when the velocity has changed to less than the nominal min value ($nominal(f) - 2.5$ knots), 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 $\delta = 100\,\mathrm{ms}$ 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 **AG** *safe*.

5.2 Experimental Results

Our 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. 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]

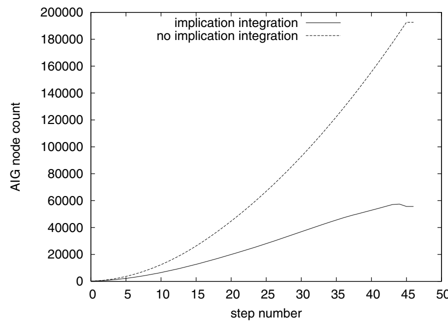

Fig. 3. Number of AIG nodes

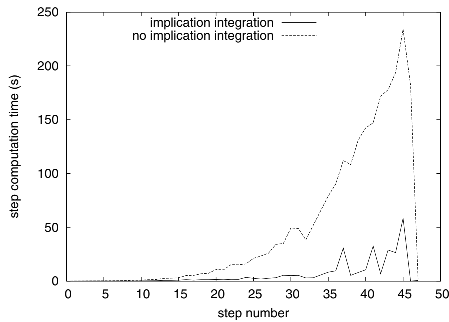

Fig. 4. CPU times for different steps

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 3 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

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

linear constraints into account. Figure 4 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).)

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 represent 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 problem.

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 Conclusions and Future Work

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.

So far, the preliminary implementation of our approach has been used to check an industrial case study with limited size and several small examples. In the future we will apply our approach to more sophisticated examples for further evaluation and for comparisons with other tools (see Sect. 6). Moreover, it seems that an integration of predicate abstraction to derive a finite-state abstraction of the hybrid system either on-the-fly or at a separate initial abstraction step (as in [3,2]) can be achieved without much difficulties. We expect that for larger examples the exectution time of our approach will heavily rely on time discretization. For this reason, 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 *Proc. CAV 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 *Proc. FST&TCS 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 *Proc. 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 *Proc. LICS 1990*, pp. 428–439.
6. A. Chutinan and B. H. Krogh. Computing polyhedral approximations to flow pipes for dynamic systems. In *Proc. IEEE CDC 1998*.
7. A. Chutinan and B. H. Krogh. Verification of the polyhedral-invariant hybrid automata using polygonal flowpipe approximations. In *Proc. HSCC 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):583–604, 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 *Proc. HSCC 2005*, *LNCS 3414*, pp. 258–273. Springer.
11. T. A. Henzinger. The theory of hybrid automata. In *Proc. LICS 1996*, pp. 278–292.
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):110–122, 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 *Proc. CHARME 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 *Proc. CAV 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 *Proc. FMCAD 2006*.
18. B. I. Silva and B. H. Krogh. Modeling and verification of hybrid system with clocked and unclocked events. In *Proc. IEEE CDC 2001*.
19. B. I. Silva, K. Richeson, B. H. Krogh, and A. Chutinan. Modeling and verification of hybrid dynamical system using CheckMate. In *Proc. 4th Conference on Automation of Mixed Processes*, 2000.
20. A. Tiwari and G. Khanna. Series of the abstractions for hybrid automata. In *Proc. HSCC 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.