Publication VI

Alessandro Cimatti and Jori Dubrovin and Tommi Junttila and Marco Roveri. Structure-Aware Computation of Predicate Abstraction. In *Formal Methods in Computer Aided Design, 9th International Conference (FMCAD 2009)*, pages 9–16, November 2009.

Structure-Aware Computation of Predicate Abstraction

Alessandro Cimatti*, Jori Dubrovin†, Tommi Junttila†, Marco Roveri*
*FBK-irst, Embedded Systems Unit, Via Sommarive 18, I-38123 Povo, Trento, Italy
†Helsinki University of Technology TKK, P.O. Box 5400, FI-02015 TKK, Finland

***Abstract*—The precise computation of abstractions is a bottleneck in many approaches to CEGAR-based verification. In this paper, we propose a novel approach, based on the use of structural information.**

Rather than computing the abstraction as a single, monolithic quantification, we provide a *structure-aware* abstraction algorithm, based on two complementary steps. The first, high-level step exploits the structure of the system, and partitions the abstraction problem into the combination of several smaller abstraction problems. This is represented as a formula with quantifiers. The second, low-level step exploits the structure of the formula, in particular the occurrence of variables within the quantifiers, and applies a set of low-level rewriting rules aiming at further reducing the scope of quantifiers.

We experimentally evaluate the approach on a substantial set of benchmarks, and show significant speed ups compared to monolithic abstraction algorithms.

I. Introduction

Many recent approaches to verification are based on Counter-Example Guided Abstraction Refinement (CEGAR). The idea is that the concrete system is overapproximated by an abstract system, that can hopefully be analyzed more easily. If the abstract system is safe, then so is the concrete system. Otherwise, the counterexample witnessing the violation in the abstract space is mapped back to the concrete space. If this operation succeeds, then there is evidence that the property is violated in the concrete space; if not, the abstract counterexample is spurious, and it can be analyzed in order to obtain indications on how to refine the abstraction. The CEGAR framework, originally proposed in the purely Boolean setting [1], has more recently been extended to the case of system representable in more expressive theories [2], [3].

Predicate abstraction is a way to guarantee conservativeness by construction [4]; basically, each predicate characterizes a set of concrete states; the abstract space is generated by collapsing in the same abstract state all the concrete states that share the same evaluation of the predicates. Some recent approaches obtained increased performance by leveraging the power of Satisfiability Modulo Theory (SMT) technologies [5], [6], [7]. Unfortunately, the computation of predicate abstractions remains a bottleneck.

Several CEGAR loops try to mitigate the problem by using imprecise predicate abstractions [2], where the transition relation of the abstract system contains spurious transitions. Although this approach eases the cost of computing the abstraction, it often results in additional CEGAR loop iterations, whose only purpose is to rule out the imprecision of the abstraction.

In this paper, we take on the problem of computing exact predicate abstractions efficiently. We consider that the methods for generating a precise predicate abstraction [5], [6], [7] are monolithic, i.e. start from a symbolic encoding, without taking into account any available structure, e.g. the possible partitioning of the concrete transition relation, the structure of predicates, or the scope of variables. We propose a method to compute predicate abstractions that is aware of, and exploits, the available structural information, thus following a divide-and-conquer approach.

The method proceeds at two different levels. At a *higher level*, we assume that the concrete system to be abstracted is described in a structured language. We discuss basic simplification principles to partition the construction of the abstraction. We instantiate the technique to the case of hybrid automata, that includes partitioning based on asynchrony, scoping of variables, handshake synchronization, and global synchronization (with timed transitions). These simplifications exploit the frame conditions (i.e. that certain variables are not modified), and the cone of influence of the predicates, trying to reduce the number of quantified variables. The technique works on the formula of the transition relation, not committing to a single formalism for models.

At a *lower level*, we transform the quantification tree by means of syntactic transformations aiming at reducing the scope of quantifications even further. This set of reductions exploits the structure of the formula to be quantified (rather than the structure of the models), and can strengthen the partitioning resulting from the high level analysis, by detecting further simplifications that are not readily visible from the structure of the input. The description of the abstract space resulting from the transformations is also structured (e.g. disjunctively partitioned), which can be used to optimize the search in the abstract space. From the technical point of view, there are several ideas that contribute to the efficiency of the approach. First, we perform iterative steps in inlining of equalities and values to simplify the formula to be quantified, and to precompute as much as possible the values of abstract variables. The second idea is to aggressively block partial results from one quantification to the next, thus focusing the search on the unexplored parts of the abstract transition relation. The third and perhaps more interesting idea, called "variable sampling", tries to split a monolithic quantification

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

even further. Variable sampling applies to the case where individual variables would be uncorrelated, were it not for the presence of a single variable: the idea is then to iteratively pick suitable values for the variable to be sampled, and to solve the resulting quantification, that can now be split into separated partitions, and separately quantified.

We implemented the proposed techniques within the NuSMV model checker [8], and we carried out an extensive experimental evaluation on benchmarks from several sources. The results show that the idea of partitioning the computation of predicate abstractions does pay off, and the proposed optimizations are effective in partitioning and in reducing the overall computation time.

The paper is structured as follows. In Sec. II, we discuss some background. In Sec. III, we present the high level structural abstractions. In Sec. IV, we discuss the low level structural abstraction algorithms. In Sec. V, we discuss some relevant related works. In Sec. VI, we report on the experimental evaluation of the approach. Finally, in Sec. VII, we draw some conclusions and outline directions for future work.

II. Background

In this section, we first define our reference formalism, based on Linear Hybrid Automata (Sec. II-A) [9], [10]. Then, in Sec. II-B, we discuss how Linear Hybrid Automata (LHA) can be represented symbolically, in a formalism that is amenable for SMT reasoning. In Sec. II-C, we define the problem of predicate abstraction for LHA. We remark that LHA are chosen as a paradigmatic formalism, generic and expressive, featuring asynchrony, synchronization, time and data variables with frame conditions. However, the results are not limited to this formalism.

A. Linear Hybrid Automata

The following definitions are based on [11], [12], [13]. A *linear atom* over a vector $\vec{X} = \langle x_1, \ldots, x_n \rangle$ of real-valued variables is an (in)equality of the form $\sum_{1 \leq i \leq n} c_i \cdot x_i \bowtie d$, where $c_1, \ldots, c_n, d \in \mathbb{Q}$ and $\bowtie \in \{<, \leq, =\}$. A *linear predicate* is a finite Boolean combination of linear atoms; a *convex linear predicate* is a finite conjunction of linear atoms. Given a vector $\vec{X} = \langle x_1, \ldots, x_n \rangle$ of variables, we write $\vec{X}'$ for $\langle x'_1, \ldots, x'_n \rangle$ and, if ϕ is an atom, predicate, or formula over $\vec{X}$, then ϕ' is obtained from ϕ by substituting each variable occurrence x_i with x'_i. If x is a variable, we denote with $\dot{x}$ the variable representing the first derivative of x w.r.t. time elapse.

A *linear hybrid automaton* is a tuple $H = \langle L, \vec{X}, \Sigma, T, Inv, Flow, Init \rangle$ consisting of the following components.

- A finite set L of *control locations*.
- A finite vector $\vec{X} = \langle x_1, \ldots, x_n \rangle$ of real-valued *data variables*.
- A finite set Σ of *synchronization labels*, not including the symbol τ for non-synchronizing transitions.
- A finite set T of *transitions*. Each transition $t \in T$ is a tuple $\langle l, \sigma, act, l' \rangle$, where $l \in L$ is the *source* location, $\sigma \in \Sigma \cup \{\tau\}$ is the *label*, act is the *action*, and $l' \in L$ is the *target* location of t. An action is a pair $act = \langle \vec{Y}, \alpha \rangle$, where $\vec{Y} \subseteq \vec{X}$ is the subset of variables that is updated when t is executed, and the linear predicate α over $\vec{X} \cup \vec{Y}'$ relates the values of the updated variables in the next state with the current values of the variables. The *closure* of act, denoted by $Clos(act)$, is the linear predicate over $\vec{X} \cup \vec{X}'$ obtained from α by adding the frame axioms for the variables that are not updated, i.e. $Clos(act) := \alpha \wedge \bigwedge_{x \in \vec{X} \setminus \vec{Y}} (x' = x)$.
- A mapping Inv from each location $l \in L$ to the *location invariant* $Inv(l)$ that is a convex linear predicate over $\vec{X}$.
- A function $Flow$ mapping each location $l \in L$ to a convex linear *flow* predicate $Flow(l)$ over $\dot{\vec{X}} = \langle \dot{x}_1, \ldots, \dot{x}_n \rangle$. The predicate $Flow(l)$ defines the allowed change derivatives for the variable values when time elapses but the automaton does not perform any discrete transition. Given a predicate $Flow(l)$, we define the predicate $Flow^\star(l)$ over $\vec{X} \cup \{\delta\} \cup \vec{X}'$, where δ is a real-valued variable, by substituting each linear atom $\sum_{1 \leq i \leq n} c_i \cdot \dot{x}_i \bowtie d$ in $Flow(l)$ by $\sum_{1 \leq i \leq n} c_i \cdot (x'_i - x_i) \bowtie d \cdot \delta$. This predicate relates the current variable values with the values after the time has elapsed the amount δ.
- The initial configuration $Init = \langle l^0, \beta \rangle$, where $l^0 \in L$ and β is a convex linear predicate over $\vec{X}$.

A *state* of the automaton H is a pair $s = \langle l, \vec{v} \rangle$, where $l \in L$ is the current control location and $\vec{v} = \langle v_1, ..., v_n \rangle \in \mathbb{R}^n$ associates each data variable x_i with a value v_i. The set of all states is denoted by S. A state $\langle l, \vec{v} \rangle$ is initial if (i) $l = l^0$ and (ii) both $\beta(\vec{v})$ and $Inv(l)(\vec{v})$ evaluate to true. The behavior of H is defined by the *transition relation* $\rightarrow_H \subseteq S \times S$ such that $\langle l_A, \vec{v}_A \rangle \rightarrow_H \langle l_B, \vec{v}_B \rangle$ if and only if (i) $Inv(l_A)(\vec{v}_A)$ and $Inv(l_B)(\vec{v}_B)$ hold, and (ii) either

- (*time elapse step*) (i) $l_B = l_A$, and (ii) $Flow^\star(l)(\vec{v}_A, \delta, \vec{v}_B)$ holds for some $\delta \geq 0$, or
- (*discrete step*) there is a transition $t = \langle l_A, \sigma, act, l_B \rangle \in T$ such that $Clos(act)(\vec{v}_A, \vec{v}_B)$ holds.

Given two hybrid automata, $H_1 = \langle L_1, \vec{X}, \Sigma_1, T_1, Inv_1, Flow_1, Init_1 \rangle$ and $H_2 = \langle L_2, \vec{X}, \Sigma_2, T_2, Inv_2, Flow_2, Init_2 \rangle$, over a common set $\vec{X}$ of variables, their *parallel composition* is the hybrid automaton

$$H_1 \otimes H_2 = \langle L_1 \times L_2, \vec{X}, \Sigma_1 \cup \Sigma_2, T, Inv, Flow, Init \rangle$$

such that

- $\langle \langle l_1, l_2 \rangle, \sigma, act, \langle l'_1, l'_2 \rangle \rangle \in T$ if and only if
  1) $\langle l_1, \sigma, act_1, l'_1 \rangle \in T_1$, $\sigma \notin \Sigma_2$, $l_2 = l'_2$, and $act = act_1$, meaning that if the automaton H_1 executes a σ-transition such that σ is either τ or not in the synchronization alphabet of H_2, then H_2 does nothing;
  2) $\langle l_2, \sigma, act_2, l'_2 \rangle \in T_2$, $\sigma \notin \Sigma_1$, $l_1 = l'_1$, and $act = act_2$, i.e. the case symmetric to the previous one;
  3) $\sigma \neq \tau$, $\langle l_1, \sigma, act_1, l'_1 \rangle \in T_1$ with $act_1 = \langle \vec{Y}_1, \alpha_1 \rangle$, $\langle l_2, \sigma, act_2, l'_2 \rangle \in T_2$ with $act_2 = \langle \vec{Y}_2, \alpha_2 \rangle$, and

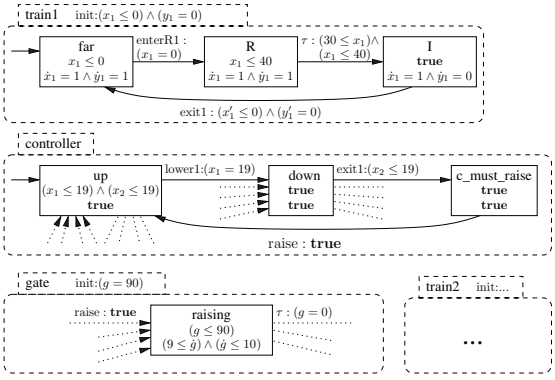

Fig. 1. A (partial) network of linear hybrid automata.

$act = \langle \vec{Y}_1 \cup \vec{Y}_2, \alpha_1 \wedge \alpha_2 \rangle$, meaning that both automata execute a σ-transition synchronously.

- $Inv(\langle l_1, l_2 \rangle) = Inv_1(l_1) \wedge Inv_2(l_2)$ for each $\langle l_1, l_2 \rangle \in L_1 \times L_2$.
- $Flow(\langle l_1, l_2 \rangle) = Flow_1(l_1) \wedge Flow_2(l_2)$ for each $\langle l_1, l_2 \rangle \in L_1 \times L_2$.
- $Init = \langle \langle l_1^0, l_2^0 \rangle, \beta_1 \wedge \beta_2 \rangle$ when $Init_1 = \langle l_1^0, \beta_1 \rangle$ and $Init_2 = \langle l_2^0, \beta_2 \rangle$.

As an example, consider the hybrid automata network "grcver" (taken from the HyTech distribution [14]) partially shown in Fig. 1. The location "far" in the automaton "train1" has the location invariant ($x_1 \le 0$) and the flow predicate ($\dot{x}_1 = 1$) ∧ ($\dot{y}_1 = 1$). In the state in which "train1" is in location "I", "controller" is in "down" and the variable x_2 has value 10, the two automata can synchronize with the label "exit1" and move to locations "far" and "c_must_raise", respectively; the other automata, which are not synchronizing with the label "exit1", do nothing.

B. Symbolic Representation

In order to apply symbolic model checking to a finite network $H_1, ..., H_k$ of linear hybrid automata, we represent the transition relation $\rightarrow_H$ for the composite automaton $H = H_1 \otimes \ldots \otimes H_k$ symbolically without explicitly constructing H. First, assume that each component automaton is of form $H_a = \langle L_a, \vec{X}, \Sigma_a, T_a, Inv_a, Flow_a, Init_a \rangle$ and define the set of *encoding variables* as $\vec{V} := \vec{X} \cup \{loc_1, ..., loc_k\} \cup \Lambda$, where each loc_a is a variable with the domain L_a used to represent the current location of the automaton H_a, and Λ is a set of *auxiliary encoding variables* introduced to encode whether a synchronizing transition is executed or not (see later for details). Now the *symbolic transition relation* encoding for H is a formula R over $\vec{V} \cup \{\delta\} \cup \vec{V}'$ such that a valuation ρ for $\vec{V} \cup \{\delta\} \cup \vec{V}'$ evaluates R to true iff the transition relation $\rightarrow_H$ has a step from the state $\langle \langle \rho(loc_1), \ldots, \rho(loc_k) \rangle, \langle \rho(x_1), ..., \rho(x_n) \rangle \rangle$ to the state $\langle \langle \rho(loc_1'), \ldots, \rho(loc_k') \rangle, \langle \rho(x_1'), ..., \rho(x_n') \rangle \rangle$. We define one such R by considering τ-transitions, synchronizing transitions, and time elapse steps separately. To simplify the presentation, we use $\Sigma_\cup$ to denote the set $\cup_{a=1}^k \Sigma_a$ of all synchronization labels and $T_{a,\sigma} = \{t \in T_a \mid t = \langle l, \sigma, act, l' \rangle\}$, where $\sigma \in \Sigma_a \cup \{\tau\}$, for the set of σ-transitions in H_a.

a) Local non-synchronizing transitions: For each component automaton H_a and for each τ-transition $t = \langle l, \tau, act, l' \rangle \in T_{a,\tau}$ in H_a, we define the encoding

$$E_t \;:=\; (loc_a = l) \wedge Clos(act) \wedge (loc_a' = l') \wedge \bigwedge_{1 \le j \le k, j \ne a} (loc_j' = loc_j). \quad (1)$$

capturing the effect of H_a executing t and stating that the other automata do nothing.

b) Synchronizing transitions: The discrete steps corresponding to the execution of synchronizing transitions in several automata is perhaps the most cumbersome to encode. Let $\sigma \in \Sigma_\cup$ be a synchronization label. Let $J_\sigma = \{a \in \{1, \ldots, k\} \mid \sigma \in \Sigma_a\}$ be the set of indices of automata in whose label set σ occurs and define the complement of J_σ by $\bar{J}_\sigma = \{1, \ldots, k\} \setminus J_\sigma$.

First, the set of encoding variables is extended by having the set Λ of auxiliary encoding variables to include a variable $Fire_{a,\sigma}$ with the domain $T_{a,\sigma}$ for each component automaton H_a. Intuitively, $Fire_{a,\sigma} = t$ if the σ-transition t is executed.

Given a σ-transition $t = \langle l, \sigma, act, l' \rangle$ with $act = \langle \vec{Y}, \alpha \rangle$ in an automaton H_a, let $Updated(t) = \vec{Y}$ and define

$$\Psi_{a,t} \;:=\; (Fire_{a,\sigma} = t) \Rightarrow (loc_a = l) \wedge \alpha \wedge (loc_a' = l'). \quad (2)$$

In addition, we define a predicate $Frame_{x,\sigma}$ that evaluates to true iff the frame conditions for the variable x are met, i.e. either some σ-transition that updates x is fired or the variable keeps its current value. Formally,

$$Frame_{x,\sigma} := (x' = x) \vee \bigvee_{a \in J_\sigma} \bigvee_{t \in T_{a,\sigma} \wedge x \in Updated(t)} (Fire_{a,\sigma} = t). \quad (3)$$

Note that if no σ-transition updates x, then $Frame_{x,\sigma} := (x' = x)$. Now define the actual encoding by

$$E_\sigma := \Big(\bigwedge_{j \in \bar{J}_\sigma} (loc_j' = loc_j)\Big) \wedge \Big(\bigwedge_{x \in \vec{X}} Frame_{x,\sigma}\Big) \wedge \Big(\bigwedge_{a \in J_\sigma} \bigwedge_{t \in T_{a,\sigma}} \Psi_{a,t}\Big) \quad (4)$$

c) Time elapse steps: In time elapse steps the component automata do not change their current locations but only time passes in a way that respects the flow predicates of the current locations of the automata. This can be captured by the encoding

$$E_\delta := (\delta > 0) \wedge \bigwedge_{1 \le a \le k} \Big((loc_a' = loc_a) \wedge \bigwedge_{l \in L_a} \Psi_{a,l}\Big) \quad (5)$$

where $\Psi_{a,l} := \big((loc_a = l) \Rightarrow Flow_a^\star(l)\big)$.

d) Putting it all together: We also define *location invariant constraint*

$$C_{\text{Inv}} := \bigwedge_{1 \le a \le k} \bigwedge_{l \in L_a} \big((loc_a = l) \Rightarrow Inv_a(l)\big) \quad (6)$$

that ensures that the location invariants of all automata hold in the current state.

Building on the components described above, we can define a compact *symbolic transition relation* for the composite automaton $H_1 \otimes \ldots \otimes H_k$ as

$$R := \Big(\bigvee_{1 \le a \le k} \bigvee_{t \in T_{a,\tau}} E_t \vee \bigvee_{\sigma \in \Sigma_\cup} E_\sigma \vee E_\delta \Big) \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \quad (7)$$

C. Computing Predicate Abstractions

Let a set $\Gamma = \{\gamma_1, \ldots, \gamma_m\}$ of predicates be given. Γ induces a partition on the set of states of the concrete system, each partition containing all the states that assign the same truth values to each predicate. Predicate abstraction defines an abstract system having as states the (finite) set of truth assignments to Γ; a transition between two abstract states a_i and a_j is possible iff there exist two concrete states s_i and s_j such that the evaluation of the predicates in s_i is a_i, the evaluation in s_j is a_j, and $s_i \rightarrow_H s_j$.

Predicate abstraction can be symbolically represented by associating to each predicate a corresponding Boolean variable. Based on the corresponding vector $\vec{P} = \langle p_1, \ldots, p_m \rangle$, we define the *abstraction constraint*

$$C_\Gamma := \bigwedge_{1 \le j \le m} (p_j \Leftrightarrow \gamma_j) \wedge (p'_j \Leftrightarrow \gamma'_j). \quad (8)$$

The *abstract transition relation* of the system obtained by applying predicate abstraction to the given system is symbolically represented by a Boolean formula $\tilde{R}$ over $\vec{P} \cup \vec{P}'$. The formula is equivalent to the following definition:

$$\tilde{R} := \exists \vec{V}, \delta, \vec{V}' : R \wedge C_\Gamma. \quad (9)$$

Similar considerations apply to the symbolic representation of the initial states, and to the abstraction of the error states. How to efficiently compute a quantifier-free Boolean presentation of the above formula is the subject of the next sections.

III. High-level Structural Abstraction

Our goal is to compute a quantifier-free presentation for the *abstract transition relation* $\tilde{R}$ over $\vec{P} \cup \vec{P}'$ defined by

$$\tilde{R} := \exists \vec{V}, \delta, \vec{V}' : R \wedge C_\Gamma.$$

Because existential quantification does not distribute over the conjunctions in (9) and (7), the computation of $\tilde{R}$ can be very expensive. However, when we push the invariant and abstraction constraints in, and distribute the quantifier over the resulting disjunctions, we obtain the *disjunctive abstract transition relation* formulation

$$\begin{aligned} \tilde{R}_{\text{disj}} := \bigvee_{1 \le a \le k} \bigvee_{t \in T_{a,\tau}} \Big(\exists \vec{V}, \delta, \vec{V}' : E_t \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma \Big) \vee \\ \bigvee_{\sigma \in \Sigma_\cup} \Big(\exists \vec{V}, \delta, \vec{V}' : E_\sigma \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma \Big) \vee \\ \Big(\exists \vec{V}, \delta, \vec{V}' : E_\delta \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma \Big). \end{aligned} \quad (10)$$

While increasing the total formula size, this enables us to decompose the computation of the abstract transition relation into smaller computations. Moreover, it enables the application of simplifications as the sub-problems have a simpler structure.

For instance, the disjunct encoding a local, non-synchronizing τ-transition $t = \langle l, \tau, act, l' \rangle \in T_{a,\tau}$ of a component automaton H_a is now of form

$$\begin{aligned} \exists \vec{V}, \delta, \vec{V}' : (loc_a = l) \wedge \alpha \wedge \\ \textstyle\bigwedge_{x \in \vec{X} \setminus \vec{Y}} (x' = x) \wedge (loc'_a = l') \wedge \\ \textstyle\bigwedge_{1 \le j \le k, j \ne a} (loc'_j = loc_j) \wedge \\ C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma \end{aligned} \quad (11)$$

where $\vec{Y} \subseteq \vec{X}$ is the set of data variables updated by the action act of the transition t. This formulation enables for the use of the equalities to eliminate the quantified variables loc_a, loc'_a, loc'_j for each $j \ne a$ and x' for each $x \in \vec{X} \setminus \vec{Y}$, by applying inlining (see Sec. IV-A).

Furthermore, this structure-based disjunctive formulation has the benefit of making the formula-based techniques described in the next section more efficient because the formulas bound by the quantifiers have less constraints and the constraints reflect the structure and locality of transitions. For instance, consider the system in Fig. 1 and assume that the set of abstraction predicates Γ is such that each predicate involves variables used either (i) only in the automaton "gate" or (ii) only in other automata. Now the computation of, e.g., the abstraction of the τ-transition in the automaton "train1" is partitioned into two parts automatically: (i) the effect of firing the transition w.r.t. the automata "train1", "train2", and "controller", and (ii) the effect w.r.t. "gate" (which after inlining reduces effectively to stating that "gate" stays in the same location).

Note that although structure-based disjunctive partitioning enhances locality and structure exploitation, there are two aspects that reduce transition and variable locality and thus partly necessitate the formula-level techniques in Sec. IV. First, location invariants of other component automata can refer to variables owned by an automaton and thus control its behavior. For instance, the enabledness of the τ-transition from location "R" of automaton "train1" is indirectly controlled by the automaton "controller" via the variable x_1: when "controller" is in location "up", the transition cannot be enabled. Second, the predicates can "globalize" a local variable if the variable is correlated with (compared with or assigned from/to, either directly or transitively) any non-local variable. When variables are correlated with others, the part of the CEGAR loop that deduces new predicates can introduce predicates that mix the variables, even if they are local to different automata. As an example, in Fig. 1 the variable g is local to the automaton "gate" and none of the transitions, location invariants, or flow predicates that refer to g refers to any other variable. But the time elapse steps implicitly connect the flow predicates of the automata "gate" and "train1" via the variable δ, and after executing the CEGAR loop few times a predicate "$\gamma_i := (9x_1 + g \le 261)$" can be found and added to the set of predicates. Because of this predicate, the abstraction of local transitions of "train1" cannot disregard the variables and location invariants of "gate" like in the previous paragraph.

A. Explicating Some Structural Invariants

We can make the individual disjuncts in $\tilde{R}_{\text{disj}}$ more amenable to the low-level techniques described in the next section by making some local structural invariants explicit in the formula level. In the following we illustrate some typical cases.

It is common in hybrid automaton that some variables never change value in time elapse steps. This applies not only to variables that are untimed by nature (e.g. counters, other discrete data) but also to variables that are used to remember and compute with the values of timed variables observed when executing transitions (e.g. a variable can be used to remember the oil temperature observed when the engine was started). For such a variable x_i, all the flow conditions $Flow_a(l)$ of one component automaton H_a (the one that conceptually owns the variable) are of form "$(\dot{x}_i = 0) \wedge ...$". This allows us to make the time elapse step specific invariant $(x'_i = x_i)$ explicit by rewriting the encoding disjunct $\exists \vec{V}, \delta, \vec{V}' : E_\delta \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma$ to $\exists \vec{V}, \delta, \vec{V}' : (x'_i = x_i) \wedge E_\delta \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma$. This transformation helps the low-level abstractor to remove quantified variables by applying inlining; on the pure formula level, noticing that $(x'_i = x_i)$ always holds whenever E_δ holds would not be this easy.

A similar case occurs when a variable x_i is used as an "exact clock" whose value changes linearly w.r.t. the elapsed time. As an example, x_1 is such a variable in the automaton "train1" in Fig. 1. In this case, all the flow conditions $Flow_a(l)$ of one component automaton H_a (the one that conceptually owns the clock) are of form "$(\dot{x} = c) \wedge ...$" for some fixed constants c (usually $c = 1$), allowing us to explicate the corresponding conjunct $(x'_i = x_i + c \cdot \delta)$ in the time elapse step encoding disjunct.

Furthermore, if all the σ-synchronizing transitions in one automaton include a common conjunct (e.g. a clock is always reset with $(x' = 0)$), then it can be explicated in the disjunct encoding σ-synchronization. For example, all the "exit1"-synchronizing transitions in the "train1" automaton in Fig. 1 include the conjuncts $(x'_1 \leq 0)$ and $(y'_1 = 0)$.

IV. Low-level Structural Abstraction

In this section we discuss how the structured representation obtained in previous section can be further simplified and evaluated to construct a Boolean presentation of the abstraction. The low level simplification routines presented in the following can also be used to simplify an unstructured description of a predicate abstraction problem. The idea is to transform the monolithic quantifier elimination problem into a sequence of smaller problems. Compared to the techniques described in previous section, operating on the automaton representation, here the transformation steps work at the level of the syntax of the formula. The simplifications are thus independent of the input formalism, and are possibly able to exploit additional structure of the original model by making cheap syntactic transformations.

For an expression β, let $vars(\beta)$ denote the set of variables occurring in β, and let $\beta[\gamma \leftarrow \alpha]$ denote a copy of β in which every occurrence of the sub-expression γ has been replaced with α.

We work on a Boolean combination of formulas of the form $\exists \vec{U} : \phi$. We assume that there are no nested quantifiers, and all free variables, that is the set $\vec{Q} := vars(\phi) \setminus \vec{U}$, are Boolean. When computing the abstract transition relation (10) of hybrid automata, we have $\vec{U} = \vec{V} \cup \vec{V}' \cup \{\delta\}$ and $\vec{Q} \subseteq \vec{P} \cup \vec{P}'$.

The simplification steps include inlining and syntactic conjunct clustering, and the idea of clustering is further generalized by a technique we call variable sampling.

A. Inlining

From the abstraction equations (10) and (11) we see that the matrix ϕ generally has the form of an n-ary conjunction. We can identify some of the conjuncts that allow simplifying the formula for less expensive quantifier elimination. The following three equivalence-preserving transformations are applied in sequence, until none of them is applicable.

1) $\exists \vec{U} : \beta \wedge (u{=}\alpha) \;\mapsto\; \exists \vec{U} : \beta[u \leftarrow \alpha]$, where $u \in \vec{U}$ and α is an expression such that $u \notin vars(\alpha)$.
2) $\exists \vec{U} : \beta \wedge (q{\Leftrightarrow}\alpha) \;\mapsto\; (q{\Leftrightarrow}\alpha) \wedge \exists \vec{U} : \beta[q \leftarrow \alpha]$, where $q \in \vec{Q}$, and α is a formula with $vars(\alpha) \subseteq \vec{Q} \setminus \{q\}$. When applying this rule, we identify the formulas q and $\neg q$ with $q \Leftrightarrow \mathbf{true}$ and $q \Leftrightarrow \mathbf{false}$ if necessary.
3) $\exists \vec{U} : \beta \wedge (\gamma{\Leftrightarrow}\alpha) \;\mapsto\; \exists \vec{U} : \beta[\gamma \leftarrow \alpha] \wedge (\gamma{\Leftrightarrow}\alpha)$, where γ and α are formulas such that $vars(\gamma) \cap \vec{U} \neq \emptyset$ and $vars(\alpha) \subseteq \vec{Q}$. This only has an effect if γ occurs as a sub-expression in β.

Transformations 1 and 2 each eliminate a variable u or q from the scope of the quantifier. Transformation 3 replaces occurrences of a formula γ with another formula α that does not contain any variables from $\vec{U}$.

As an example, consider the formula $\exists x, x' : (p_1 \Leftrightarrow (x > 0)) \wedge (p'_1 \Leftrightarrow (x' > 0)) \wedge (x' = x)$. Using Transformation 1 with $u := x$ and $\alpha := x'$, we inline the frame condition $(x' = x)$ and obtain $\exists x, x' : (p_1 \Leftrightarrow (x > 0)) \wedge (p'_1 \Leftrightarrow (x > 0))$. Then, using Transformations 3 and 2 in this order, we get first $\exists x, x' : (p_1 \Leftrightarrow (x > 0)) \wedge (p'_1 \Leftrightarrow p_1)$ and then $(p'_1 \Leftrightarrow p_1) \wedge \exists x, x' : (p_1 \Leftrightarrow (x > 0))$. As a result, we have obtained a frame condition for p_1 and eliminated p'_1 and x' from the scope of the quantifier.

B. Syntactic Conjunct Clustering

From the quantifier elimination problem $\exists \vec{U} : \phi$, where ϕ is an n-ary conjunction, we can identify clusters of conjuncts such that each variable of $\vec{U}$ occurs in at most one of them. Let us rearrange ϕ into conjuncts $\phi_0 \wedge \phi_1 \wedge \cdots \wedge \phi_n$ such that ϕ_0 contains no variables from $\vec{U}$, and for all $1 \leq i < j \leq n$, the sets $vars(\phi_i) \cap \vec{U}$ and $vars(\phi_j) \cap \vec{U}$ are disjoint. Then, $\exists \vec{U} : \phi$ is replaced with the equivalent partitioned formula $\phi_0 \wedge (\exists \vec{U} : \phi_1) \wedge \cdots \wedge (\exists \vec{U} : \phi_n)$. This enables solving a sequence of smaller quantifier elimination problems instead of one large one even when the matrix ϕ is a conjunction.

C. Variable Sampling

We are often able to benefit from a conjunctive partitioning even if the conjuncts are not totally disjoint w.r.t. the set $\vec{U}$. This is particularly evident in the case of time elapse steps, where the δ variable occurs in the same linear atom with many or all data variables, preventing syntactic clustering. We notice, however, that when δ is instantiated to a specific numerical value, clustering wrt. the remaining variables may become possible. In the general setting, assume that $\phi \equiv \phi_1 \wedge \cdots \wedge \phi_n$ and there is a subset $\vec{W} \subseteq \vec{U}$ such that for all $1 \leq i < j \leq n$, the set $vars(\phi_i) \cap vars(\phi_j) \cap \vec{U}$ is a subset of $\vec{W}$. Then, $\exists \vec{U} : \phi$ is equivalent to $\exists \vec{W} : \big((\exists \vec{U} \setminus \vec{W} : \phi_1) \wedge \cdots \wedge (\exists \vec{U} \setminus \vec{W} : \phi_n)\big)$. The sub-problems $\exists \vec{U} \setminus \vec{W} : \phi_i$ cannot be solved directly using efficient SMT-based techniques [6], [7] when $\vec{W}$ contains non-Boolean variables. However, if the values of all variables in $\vec{W}$ are fixed, then the free variables in the sub-problems are all Boolean, and the sub-problems can be solved in sequence. By performing the quantifier elimination when $\vec{W}$ is fixed, we get an under-approximation of $\exists \vec{U} : \phi$. The following procedure shows that by sampling only a finite number of valuations of $\vec{W}$, we can accumulate the precise formula of $\exists \vec{U} : \phi$, which is then returned by the procedure.

1) $\alpha \leftarrow \mathbf{false}$
2) **while** $\phi \wedge \neg\alpha$ is satisfiable:
3) $\quad w \leftarrow$ a valuation of $\vec{W}$ that satisfies $\phi \wedge \neg\alpha$
4) $\quad \alpha \leftarrow \alpha \vee \mathrm{Elim}\big(\neg\alpha \wedge \bigwedge_{1 \leq i \leq n} \exists \vec{U} : \phi_i[\vec{W} \leftarrow w]\big)$
5) **end while**
6) **return** α

On line 3, an SMT solver is used to discover a new valuation of $\vec{W}$. On line 4, the expression $\phi_i[\vec{W} \leftarrow w]$ denotes the assignment of the values w to $\vec{W}$ in ϕ_i, and Elim is a procedure that eliminates the quantifiers from its argument and returns the resulting formula over $\vec{Q}$. The benefit is that Elim can now exploit syntactic conjunct clustering and eliminate the quantifiers in sequence.

The above procedure maintains the invariant that α is an under-approximation of $\exists \vec{U} : \phi$. On the other hand, α is equivalent to $\exists \vec{U} : \phi$ when the formula $\phi \wedge \neg\alpha$ is unsatisfiable, and this is when the procedure returns. Termination of the loop follows from the fact that on line 4, procedure Elim always finds at least one new valuation of $\vec{Q}$ that makes α false and $\exists \vec{U} : \phi$ true, and there is only a finite number of possible valuations of the Boolean variables $\vec{Q}$.

In abstraction problems originating from hybrid systems, the set $\{\delta\}$ is a good candidate for the set $\vec{W}$. Effectively, this means computing the abstraction of the time elapse step in pieces, where each piece corresponds to a fixed value of δ, i.e. a fixed-length time interval.

D. Blocking Visited Models

Once the simplifications have been carried out, quantifier elimination to the remaining parts is applied. We remark that, when computing a disjunction of quantifications, it is possible to restrict the search to the negation of the models computed so far. This can pushed even further by considering that at a certain point of the abstraction computation we may end up with a formula of the form $\alpha \vee (\gamma \wedge \exists \vec{U} : \phi)$, where the subformulas α and γ do not contain quantifiers or variables from $\vec{U}$. A valuation of the free variables $vars(\phi) \setminus \vec{U}$ in $\exists \vec{U} : \phi$ is a don't care if it entails α or $\neg\gamma$. Thus, we can further reduce the models the quantifier elimination has to enumerate by computing the Boolean formula $\exists \vec{U} : (\gamma \wedge \neg\alpha \wedge \phi)$.

V. Related Work

In the Boolean setting, quantification is typically used for the basic operation of Symbolic Model Checking, i.e. image computation. Quantification procedures based on Binary Decision Diagrams (BDDs) have been aggressively optimized [15], [16], [17]. Depending on the nature of the design under verification, the corresponding transition relation was disjunctively or conjunctively decomposed, and quantifiers were pushed inside as to operate on possibly smaller BDDs. The above transformations typically operate on the set of support of BDDs. More recently, SAT-based quantifier elimination has been investigated [18], [19], also in combination with BDD-based techniques [20].

The preliminary idea of pushing quantifiers inside of conjuncts and disjuncts for computing abstraction has been firstly discussed in [21]. However, although the description was done in a general settings, it was applied for abstracting finite domains using BDD operations.

The idea of using decision procedures for computing abstractions has been explored in [22], [23]. The work in [6] improves over them by lifting DPLL-based quantification to the case of SMT. The approach, referred to as All-SMT, is based on the use of an SMT solver, which iteratively finds models that satisfy the formula under the quantifier; each satisfying assignment of the free (Boolean) variables is added as a blocking clause, and the search is restarted until no more models are found. The work in [7] attempts to overcome some of the inefficiencies of [6] by combining BDD-based reasoning and SMT techniques. Compared to the work presented in this paper, both [6] and [7] tackle the problem of predicate abstraction as a monolithic problem. We remark that, any of these two techniques can be used as back-ends in the procedure presented in this paper.

Several approaches trade precision for accuracy. In fact, [6] also shows how to approximate the results. A similar line is followed in [2], [24], where different approximate methods for the computation of predicate abstractions are presented. The main problem is that approximation in the abstraction may lead to additional iterations in the CEGAR loop. In this paper, we concentrate on the computation of the exact abstraction for a given set of predicates.

Several approaches to software model checking rely on the construction of an abstract space based on the availability of predicates [25], [26]. In a sense, the approach is exploiting the structure of the control flow graph to partition the problem of computing the abstractions: the abstract system shares the same control flow graph with the program being abstracted. The main difference with our approach is in that here we do not have a single sequential program, but rather a set of

concurrent programs interacting via shared variables and with global timed transitions, which poses additional difficulties.

Several notable abstraction mechanisms have been proposed in settings other than predicate abstraction. The work by Segelken [27] addresses abstraction for concurrent systems by retaining the control structure of the automata being analyzed but dropping information and over-approximating the transition relation. A similar approach is also used in [28] for timed systems, and [29], [30] for hybrid systems.

VI. Implementation and Experiments

We have implemented the proposed techniques within an extended version of the NuSMV model checker [8] that allows for variables of type Real and is connected to the MathSAT SMT engine [31]. This version allows for bounded model checking, and a CEGAR loop based on predicate abstraction. The proposed techniques have been implemented as follows. The networks of automata have been described as data structures in the Python programming language. The high level abstraction is implemented as a Python front-end that generates from the network of automata an abstraction problem as a formula either in monolithic form $\tilde{R}$ or disjunctive form $\tilde{R}_{\text{disj}}$. The lower level computation engine is implemented within NuSMV, and relies on the formula manipulation routines to implement inlining and syntactic conjunct clustering. The quantifier elimination can either be performed by the All-SMT functionalities provided by MathSAT, or by hybrid techniques combining BDDs and SMT [7]. The variable sampling procedure is obtained by integration with MathSAT. In the implementation we exploit the incrementality provided by MathSAT, so that learnt theory lemmas can be reused in the different calls.

For the experimental evaluation, we considered two sets of benchmarks. The first set is taken from the examples distributed together with HyTech [14]. The predicates for the abstraction of this set were found automatically using the CEGAR loop in NuSMV while proving/disproving the property. The second set is obtained by means of a script that generates structured descriptions for networks of linear hybrid automata; the examples contain non-synchronizing and synchronizing transitions and flow constraints, and data variables with a controlled amount of interdependencies. Each predicate for abstraction corresponds to either a location of an automaton being active or to a random linear atom being true. In the experiments we varied the number of parallel automata between 3 and 5, predicates between 13 and 37, data variables between 3 and 39, and top-level disjuncts between 18 and 38. The set contains 756 different models in total.

The problem we consider consists of the generation of the (BDD representing the) transition relation of the system obtained by precise abstraction with respect to the set of predicates. In the experiments we disregard the computation of abstract initial state and property formulas, since they are in general much easier to obtain than the transition relation.

For each of the benchmarks, we compare the proposed abstraction techniques to the direct monolithic quantifier elimination. When moving from monolithic to structural abstraction, we incrementally enable the different techniques to assess their effectiveness individually. In the experiments, we use the MathSAT All-SMT quantifier elimination procedure. The hybrid techniques of [7] shows similar trends. The benchmarks were run on a cluster of Intel Xeon 5130 based machines, with one CPU core allocated for each problem. In all runs, we used a time out of 10 minutes and a memory limit of 1GB. The software and models used in the experimental evaluation are available at http://es.fbk.eu/people/roveri/tests/fmcad09/.

TABLE I
Abstraction results for HyTech models.

Model	$\lvert\vec{P}\rvert$	$\lvert\vec{V}\rvert$	$\lvert$disj$\rvert$	computation time (s) monol.	partit.	clust.	sampl.	sampling clu	sam
active	34	5	27	54.626	18.847	2.410	0.937	5	1
active-trace	34	7	27	51.781	22.171	2.473	0.952	5	1
audio	30	6	15	13.826	4.547	0.448	0.442	2	2
audio-timing	29	7	15	10.910	3.915	0.947	0.690	2	6
billiard-timed	25	3	5	0.910	0.732	0.732	1.044	2	13
dist-controller	8	7	12	0.320	0.232	0.195	0.147	5	1
grc-ver	24	5	11	33.068	19.599	10.421	0.455	4	8
new-grc	22	5	11	38.649	17.840	7.395	0.383	4	7
railroad	16	3	8	0.170	0.140	0.131	0.112	2	5
reactor-clock	19	4	5	0.181	0.133	0.069	0.050	2	2
reactor-rect	17	4	5	0.132	0.112	0.051	0.045	2	2

Table I shows the results for the HyTech automata. The columns $|\vec{P}|$, $|\vec{V}|$, and |disj| show the number of predicates, the number of data variables, and the number of top-level disjuncts in $\tilde{R}_{\text{disj}}$ for each instance. The column "monol." shows the run time in seconds for computing the abstract transition relation using monolithic All-SMT on the compact non-partitioned formula $\tilde{R}$. The column "partit." shows the run time with disjunctive partitioning and inlining enabled. In the column "clust.", syntactic conjunct clustering is enabled in addition, and in the column "sampl.", also the variable sampling of Sec. IV-C is used with $\vec{W} = \{\delta\}$. The listed times are average times over 9 identical runs given a fixed set of predicates. No individual run time was more than 14% off the average. Some HyTech models that were experimented with are omitted from Table I, because in those cases the run time is less than 0.1 s using all four approaches, with no measurable differences.

Fig. 2 shows corresponding run times for the randomly generated hybrid automata. Each marker represents a network of automata and a set of predicates, for which the abstract transition relation was computed once using monolithic All-SMT (run time shown on the x-axis), once using disjunctive partitioning with inlining enabled (the y-axis of Fig. 2(a)), and once with also syntactic conjunct clustering and variable sampling wrt. the δ variable enabled (the y-axis of Fig. 2(b)).

The results clearly show that the presented techniques can dramatically speed up the abstraction computation. Moreover, the enabling of all the features together results in the biggest improvement. We remark that the variable sampling technique is able to increase the number of clusters in all of the HyTech models of Table I (this is not generally always the case), and this often makes the particularly expensive abstraction of the time elapse step much more tractable. The rightmost columns

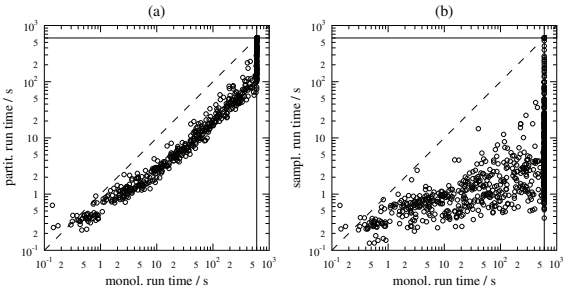

Fig. 2. Abstraction computation times for randomly generated LHA.

"clu" and "sam" show the number of clusters resulting from fixing δ and the number of values of δ that were sampled to obtain the precise abstraction. Variable sampling wrt. variables other than δ might give further speedup, but was not tested in these experiments.

VII. Conclusions and Future Work

In this paper, we have tackled the problem of computing precise predicate abstractions. We have proposed two main contributions. On the high level, we have shown that it is possible to exploit several features common in standard modeling languages in order to obtain high level simplifications. At the low level, we propose to manipulate the logical formulation of the abstraction problem with transformations such as inlining, quantification push-in, and the novel variable sampling. This divide-and-conquer approach results in simpler formulations, that, as shown by the experimental evaluation, can be more effectively computed.

The approach presented in this paper also enables for an easy use of standard optimizations for image computation in the abstract space, e.g. conjunctive and disjunctive partitioning of the transition relation [15], [16], [17].

In the future, we will investigate the application of the proposed techniques for the abstract reachability, according to the lines defined in [22], and a formulation that is able to provide incrementality for the abstraction-refinement loop.

Acknowledgements

The financial support of Helsinki Graduate School in Computer Science and Engineering, Emil Aaltonen Foundation, Academy of Finland (project 112016), and Technology Industries of Finland Centennial Foundation is gratefully acknowledged. A. Cimatti and M. Roveri are sponsored by the European Commission with project FP7-2007-IST-1-217069 COCONUT.

References

[1] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[2] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, "Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog," in *Design Automation Conference (DAC)*, June 2005.

[3] M. K. Ganai and A. Gupta, "Completeness in SMT-based BMC for software programs," in *DATE*. IEEE, 2008, pp. 831–836.

[4] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *CAV*, ser. LNCS, vol. 1254. Springer, 1997, pp. 72–83.

[5] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *The Handbook of Satisfiability*. IOS Press, 2009.

[6] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, "SMT techniques for fast predicate abstraction," in *CAV*, ser. LNCS, vol. 4144. Springer, 2006, pp. 424–437.

[7] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar, "Computing predicate abstractions by integrating BDDs and SMT solvers," in *FMCAD*. IEEE C. S., 2007, pp. 69–76.

[8] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model checker," *STTT*, vol. 2, no. 4, pp. 410–425, 2000.

[9] R. Alur, T. Dang, and F. Ivančić, "Predicate abstraction for reachability analysis of hybrid systems," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 1, pp. 152–199, 2006.

[10] ——, "Counterexample-guided predicate abstraction of hybrid systems," *Theoretical Computer Science*, vol. 354, pp. 250–271, 2006.

[11] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theor. Comp. Sci.*, vol. 138, pp. 3–34, 1995.

[12] R. Alur, T. A. Henzinger, and P.-H. Ho, "Automatic symbolic verification of embedded systems," *IEEE Trans. on Sw. Eng.*, vol. 22, no. 3, pp. 181–201, 1996.

[13] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HyTech: a model checker for hybrid systems," *STTT*, vol. 1, pp. 110–122, 1997.

[14] "HyTech: The HYbrid TECHnology tool," May 2009, http://embedded.eecs.berkeley.edu/research/hytech/.

[15] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic model checking with partitioned transition relations," in *VLSI 91*, ser. IFIP Transactions. North-Holland, 1991, pp. 49–58.

[16] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton, "Efficient BDD algorithms for FSM synthesis and verification," in *IEEE/ACM Proc. International Workshop on Logic Synthesis*, May 1995.

[17] D. Geist and I. Beer, "Efficient model checking by automated ordering of transition relation partitions," in *CAV*, ser. LNCS, no. 818. Springer, 1994, pp. 299–310.

[18] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002, pp. 250–264.

[19] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based Unbounded Symbolic Model Checking Using Circuit Cofactoring," in *ICCAD*. IEEE Computer Society / ACM, 2004, pp. 510–517.

[20] O. Grumberg, A. Schuster, and A. Yagdar, "Hybrid BDD and All-SAT Method for Model Checking," in *Symposium on Satisfiability Solvers and Program Verification (SSPV)*, Seattle, USA, Aug. 2006.

[21] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.

[22] S. K. Lahiri, R. E. Bryant, and B. Cook, "A symbolic approach to predicate abstraction," in *CAV*, ser. LNCS, vol. 2725. Springer, 2003, pp. 141–153.

[23] S. K. Lahiri, T. Ball, and B. Cook, "Predicate abstraction via symbolic decision procedures," in *CAV*, ser. LNCS, vol. 3576. Springer, 2005, pp. 24–38.

[24] D. Kroening and N. Sharygina, "Image computation and predicate refinement for RTL Verilog using word level proofs," in *DATE*. ACM, 2007, pp. 1325–1330.

[25] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*. ACM, 2002, pp. 58–70.

[26] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 570–574.

[27] M. Segelken, "Abstraction and counterexample-guided construction of ω-automata for model checking of step-discrete linear hybrid models," in *CAV 2007*, ser. LNCS, vol. 4590. Springer, 2007, pp. 433–448.

[28] S. Kemper and A. Platzer, "Sat-based abstraction refinement for real-time systems," *ENTCS*, vol. 182, pp. 107–122, 2007.

[29] A. Tiwari, "Abstractions for hybrid systems," *Formal Methods in System Design*, vol. 32, no. 1, pp. 57–83, 2008.

[30] M. Fränzle, "Verification of hybrid systems," in *CAV*, ser. LNCS, vol. 4590. Springer, 2007, p. 38.

[31] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4SMT Solver," in *CAV*, ser. LNCS, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 299–303.