HAL
open science

Discrete Controller Synthesis for Infinite State Systems with ReaX

Nicolas Berthier, Hervé Marchand

To cite this version:

Nicolas Berthier, Hervé Marchand. Discrete Controller Synthesis for Infinite State Systems with ReaX. IEEE International Workshop on Discrete Event Systems, May 2014, Cachan, France. pp.420-427. hal-00974553

HAL Id: hal-00974553

https://inria.hal.science/hal-00974553v1

Submitted on 7 Apr 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Discrete Controller Synthesis for Infinite State Systems with ReaX ⋆

Nicolas Berthier * **Hervé Marchand** *

* *INRIA Rennes - Bretagne Atlantique, Rennes, France*

Abstract: In this paper, we investigate the control of infinite reactive synchronous systems modeled by arithmetic symbolic transition systems for safety properties. We provide effective algorithms allowing to solve the safety control problem, and report on experiments based on ReaX, our tool implementing these algorithms.

Keywords: Control synthesis, automata with infinite variables, synchronous paradigm.

1. INTRODUCTION

The control theory of discrete event systems (Ramadge and Wonham (1989); Cassandras and Lafortune (2007)) allows the use of constructive methods ensuring, *a priori* and by means of control, required properties on a system's behavior. Usually, the starting point of these theories is: given a model for the system and control objectives, a controller must be derived by various means such that the resulting behavior of the closed-loop system meets the control objectives. When modeling realistic systems, it is often convenient to manipulate state/event variables instead of simply atomic states/events. Within this framework, the states (as well as the events) can be seen as instantiations of vectors of variables. In this case, usual control techniques entail instantiating the variables during state space exploration and analysis. Whereas this enumeration leads to the classical state space explosion when the variables take their values in a finite set, it may also be unfeasible whenever the domain of some variables is infinite.

Related Works The control of infinite systems have been subject to several studies. One can think about the control of Timed Automata (Cassez et al., 2005), or Vector of Discrete Events Systems (Li and Wohnam, 1994). Kumar and Garg (2005) extend their previous work (Kumar et al., 1993) to consider infinite systems. They prove that, in that case, the state avoidance control problem is undecidable. They also show that the problem can be solved in the case of Petri Nets, when the set of forbidden states *Bad* is upward-closed. The controller synthesis of *infinite state systems* modeled by Petri Nets has also been considered by Holloway et al. (1997). The control of symbolic transition systems with variables has also been studied by Le Gall et al. (2005) and Kalyon et al. (2011) in an asynchronous framework and with finite alphabets.

Here, we consider reactive systems within the synchronous framework (Benveniste et al., 2003), *i.e.,* data-flow systems reacting to inputs sent by the environment, and producing outputs resulting from internal transformations. In this framework, part of the inputs is uncontrollable (it may correspond to measures from sensors), whereas the other part of inputs is (*e.g.,* user commands to actuators, that make the system evolve from a configuration to some other one). The aim of the controller is then to reduce the possible values of the controllable inputs so as to ensure some properties. The control of finite synchronous programs handling only Boolean variables has been subject to several studies (Marchand and Samaan, 2000; Marchand et al., 2000) and there exists a tool Sigali implementing this theory. This tool is integrated in the BZR environment (Delaval et al., 2010, 2013), as part of a compiler of reactive synchronous programs with contracts.

Contributions In this paper, we assume that the systems are modeled by Arithmetic Symbolic Transition Systems (ASTSs), which are transition systems with variables (Boolean, integer, real, etc.) encoding both the inputs of the system and its internal behavior. This model has a finite structure and offers a compact way to specify systems handling data [1]. We provide algorithms allowing to compute a controller ensuring safety properties. As variables in infinite domains are manipulated, these algorithms are based on interpretation techniques that over-approximate the state space that has to be forbidden by control, hence performs the computation on an abstract (infinite) domain. Our technique is similar to the one of Le Gall et al. (2005) and Kalyon et al. (2011), who consider the control of automata with internal variables, but with a finite alphabet of actions [2]. We present the ASTS model and our algorithms in Section 2.

After having detailed some technical choices in Section 3, we present in Section 4 our implementation of these algorithms in the tool ReaX. This tool computes the maximally permissive controller whenever all variables are Boolean, and provide a correct but not necessarily maximally permissive controller whenever some variables are in infinite domains like integer of real. ReaX strictly extends the ability of Sigali (Marchand et al., 2000), that only manipulates Boolean variables (Marchand and

⋆ This work was supported by the French ANR project Ctrl-Green (ANR-11-INFR 012 11), funded by ANR INFRA and MINALOGIC.

[1] Note that Petri Nets as well as Vector of Discrete Events Systems can easily be encoded by means of ASTSs.

[2] Miremadi et al. (2008) propose a similar framework, implemented in Supremica, but the domain of the variables is assumed to be finite.

Samaan, 2000). The aim is to replace SIGALI within the compilation process of BZR, so as to be able to compute controllers handling numerical aspects when modeling systems and expressing properties. We finally present in Section 5 some examples showing the interests of our tool, and report on its preliminary effective usage through BZR with a realistic use case for the coordination of multi-tier autonomic management decisions.

2. CONTROL OF SYMBOLIC TRANSITION SYSTEMS

We shortly introduce in this section the model of Arithmetic Symbolic Transition Systems allowing to model reactive systems handling data (a mix between the STS introduced by Kalyon et al. (2011) and the Boolean system of Marchand et al. (2000)). We further formally present the invariance control problem and our algorithms within this framework.

2.1 (Controllable) Discrete Transition System

The model of Arithmetic Symbolic Transition Systems (ASTS) is a transition system with (internal or input) variables whose domain can be infinite, and composed of a finite set of symbolic transitions. Each transition is guarded on the system variables, and has an update function indicating the variable changes when a transition is fired. This model allows the representation of infinite systems whenever the variables take their values in an infinite domain, while it has a finite structure and offers a compact way to specify systems handling data.

Notations Let $V = \langle v_1, \ldots, v_n \rangle$ be a tuple of variables and $\mathcal{D}_v$ the (infinite) domain of v. We note $\mathcal{D}_V = \prod_{i \in [1,n]} \mathcal{D}_{v_i}$ the (infinite) domain of V.

Arithmetic Symbolic Transition System

Definition 1. (ASTS). *An* Arithmetic Symbolic Transition System *is a tuple $S = \langle X, I, T, A, \Theta_0 \rangle$ where:*

- *$X = \langle x_1, \ldots, x_n \rangle$ is a vector of state variables ranging over $\mathcal{D}_X = \prod_{j \in [1,n]} \mathcal{D}_{x_j}$ and encoding the memory necessary for describing the system behavior;*
- *$I = \langle i_1, \ldots, i_m \rangle$ is a vector of variables that ranges over $\mathcal{D}_I = \prod_{j \in [1,m]} \mathcal{D}_{i_j}$, called input variables;*
- *T is of the form $(x_i := T^{x_i})_{x_i \in X}$, such that, for each $x_i \in X$, the right-hand side T^{x_i} of the assignment $x_i := T^{x_i}$ is an expression on $X \cup I$. T is called the transition function of S, and encodes the evolution of the state variable x_i. It characterizes the dynamic of the system between the current state and the next state when receiving an input vector.*
- *A is a predicate with variables in $X \cup I$ encoding an assertion on the possible values of the inputs depending on the current state;*
- *Θ_0 is a predicate with variables in X encoding the set of initial states.*

For technical reasons, we shall assume that A is expressed in a theory that is closed under quantifier elimination; that is, any predicate containing quantifiers is equivalent to one without quantifiers (and over the same set of variables). For example, Presburger arithmetic with function symbols satisfies these constraints (provided that quantifiers do not bind variables within the scope of a function) and is expressive enough to express common data structures such as integers, reals, etc.

Note also that any ASTS is deterministic by definition (due to the form of T) as soon as the set of solutions of Θ_0 is reduced to a singleton.

Remark 1. We qualify as logico-numerical *an ASTS whose state and input variables are Boolean variables ($\mathbb{B}$) or numerical variables (typically, $\mathbb{R}$ or $\mathbb{Z}$), i.e., such that $X = \mathbb{B}^k \cup \mathbb{R}^{k'} \cup \mathbb{Z}^{k''}$ with $k + k' + k'' = n$ (and similarly for the input variables). In the sequel, we shall focus on this kind of systems.*

ASTSs can conveniently be represented as parallel compositions of Mealy automata with numerical variables. So, in the sequel, we shall represent our examples in this form, hence rendering their Boolean state variables implicit.

An automaton representation of such an ASTS is shown in Figure 5 of Section 5.2. In this example, the automaton implicitly embodies a single Boolean state variable (as it has two states); a is a Boolean output holding when a transition leading to Active is taken, r and s are Boolean non-controllable inputs, and c is a Boolean controllable input. The numerical variable x (initially null), is reset when Active is entered, and incremented when a transition leaving Active is taken.

To each ASTS, one can make correspond an Infinite Transition System (ITS) defined as follows:

Definition 2. (ITS). *Given an ASTS $S = \langle X, I, T, A, \Theta_0 \rangle$, we make correspond an ITS $[S] = \langle \mathcal{X}, \mathcal{I}, \mathcal{T}_S, \mathcal{A}_S, \mathcal{X}_0 \rangle$ where:*

- *$\mathcal{X} = \mathcal{D}_X$ is the state space of $[S]$;*
- *$\mathcal{X}_0 \subseteq \mathcal{X}$ is the set of initial states, and is such that*
$$\mathcal{X}_0 = \{x \in \mathcal{D}_X \mid \Theta_0(x) = \mathit{true}\};$$
- *$\mathcal{I} = \mathcal{D}_I$ is the input space of $[S]$;*
- *$\mathcal{T}_S \subseteq \mathcal{X} \times \mathcal{I} \to \mathcal{X}$ is such that*
$$\mathcal{T}_S(x, \iota) = (x'_j)_{j \in [1,n]} \Leftrightarrow \forall j \in [1,n], x'_j := T^{x_j}(x, \iota);$$
- *$\mathcal{A}_S \subseteq \mathcal{X} \times \mathcal{I}$ is such that*
$$\mathcal{A}_S = \{(x, \iota) \mid A(x, \iota) = \mathit{true}\}.$$

The behavior of such a system is as follows. $[S]$ starts in a state $x_0 \in \mathcal{X}_0$. Assuming that $[S]$ is in a state $x \in \mathcal{X}$, then upon the reception of an input $\iota \in \mathcal{I}$ such that $(x, \iota) \in \mathcal{A}_S$, $[S]$ evolves in the state $x' = \mathcal{T}_S(x, \iota)$.

We shall denote by $(x^0, \iota^0).(x^1, \iota^1).(x^2, \iota^2).\cdots$ an infinite sequence of states/inputs of $[S]$ that can be constructed following the preceding rule ($x^0 \in \mathcal{X}_0$ and $\forall j \in \mathbb{N}$, $x^{j+1} = \mathcal{T}_S(x^j, \iota^j)$ and $(x^j, \iota^j) \in \mathcal{A}_S$). We denote by $\mathit{Trace}([S])$ this set of sequences and by $\mathit{XTrace}([S])$ the sequences of states that are obtained from $\mathit{Trace}([S])$ by removing the input component of each tuple of the sequences.

Given an ASTS $S = \langle X, I, T, A, \Theta_0 \rangle$, its associated ITS $[S]$ and a predicate Φ over X, we say that S satisfies Φ (noted $S \models \Phi$) whenever $\mathit{XTrace}([S]) \subseteq \{x \in \mathcal{D}_X \mid \Phi(x) = \mathit{true}\}$.

Remark 2. Note that in the reminder of this paper, we will use alternatively S or $[S]$ depending on the context. Similarly, by abuse of notation, we will often use P to denote the predicate and its associated set of solutions.

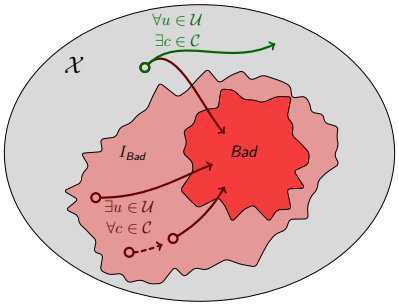

Fig. 1. Exact computation of the forbidden states I_{Bad}.

2.2 Control of an ASTS: Boolean Case

Assume given a system S and a predicate Φ on S. Our aim is to restrict the behavior of S by means of control in order to fulfill Φ.

Means of Control The control of an ASTS relies on a distinction between the inputs. We distinguish between the uncontrollable input variables I_{uc} which are defined by the environment, and the controllable input variables I_c which are defined/restricted by the controller of the system.

Note that the partitioning of the input variables in S induces a "partitioning" of the input space in $[S]$. If we denote $\mathcal{U} = \mathcal{D}_{I_{uc}}$ and $\mathcal{C} = \mathcal{D}_{I_c}$, then in $[S]$, we have $\mathcal{I} = \mathcal{U} \times \mathcal{C}$.

Within our framework, a controller is given by a predicate A_Φ over $X \cup I_{uc} \cup I_c$ that constrains the set of admissible controllable inputs so that the traces of the controlled system always satisfy Φ.

Definition 3. (Discrete Controller Synthesis Problem). *Given an ASTS $S = \langle X, I_{uc} \uplus I_c, T, A, \Theta_0 \rangle$ and a predicate Φ over X, solving the* discrete controller synthesis problem *is to compute a predicate A_Φ such that*

$$S' = \langle X, I_{uc} \uplus I_c, T, A_\Phi, \Theta_0 \rangle \models \Phi$$

and $\forall v \in X \cup I_{uc} \cup I_c, A_\Phi(v) \Rightarrow A(v)$.

Computation of the Forbidden States Within a finite setting in which $X = \mathbb{B}^n$ and $I = \mathbb{B}^m$, the solution of this control problem is well known and relies on some fix-point computation of predicate (Marchand and Samaan, 2000). When dealing with numerical variables, the algorithm is actually the same (with the difference that the fix-point computation may never terminate). We thus just recall in the sequel the principle of the controller computation and emphasize the part(s) that may not be actually computed when dealing with logico-numerical ASTSs.

We first define the set $Bad \stackrel{\text{def}}{=} \{x \in \mathcal{D}_X \mid \Phi(x) = \mathit{false}\}$, expressing, in terms of the ITS $[S]$, the set of states that do not satisfy Φ. Obtaining a controller ensuring the unreachability of Bad essentially boils down to compute the set of forbidden states I_{Bad}, that can uncontrollably lead, in any number of transitions, into a state in Bad. We illustrate this principle in Figure 1.

The definition of $\mathrm{pre}_u \colon \wp(\mathcal{X}) \to \wp(\mathcal{X})$, which computes the set of states that can uncontrollably lead in one transition to a state belonging to a given set of states, is as follows:

$$\mathrm{pre}_u(B) \stackrel{\text{def}}{=} \{x \in \mathcal{X} \mid \exists u \in \mathcal{U}, \forall c \in \mathcal{C}, (x, u, c) \in \mathcal{T}_S^{-1}(B) \cap \mathcal{A}_S\}$$

where $\mathcal{T}_S^{-1} \colon \wp(\mathcal{X}) \to \wp(\mathcal{X} \times \mathcal{U} \times \mathcal{C})$ denotes the "open" pre-image function (also giving inputs) of the ITS $[S]$.

Further, the set of forbidden states can be computed as $I_{Bad} = \mathrm{coreach}_u(Bad)$, with $\mathrm{coreach}_u \colon \wp(\mathcal{X}) \to \wp(\mathcal{X})$ defined as the least fix-point (lfp):

$$\mathrm{coreach}_u(B) \stackrel{\text{def}}{=} \mathrm{lfp}(\lambda\beta.B \cup \mathrm{pre}_u(\beta))$$

Note that the limit of the fix-point $\mathrm{coreach}_u(Bad)$ actually exists as the function $\mathrm{coreach}_u$ is monotonic, but may not be computable when dealing with numerical variables.

If $\mathcal{X}_0 \cap I_{Bad} \neq \varnothing$, then the synthesis fails (Marchand and Samaan, 2000); *i.e.,* the set of leavers given by $\mathcal{C}$ does not provide sufficient means to ensure the unreachability of Bad. Otherwise, assuming that I_{Bad} can actually be computed, one can obtain from it a new set $\mathcal{A}_\Phi \subseteq \mathcal{X} \times \mathcal{U} \times \mathcal{C}$, expressing the constraints over the state and input spaces that must be satisfied for Bad to be unreachable. It is obtained as:

$$\mathcal{A}_\Phi = \mathcal{T}_S^{-1}(I_{Bad}^c) \cap \mathcal{A}_S$$

In terms of the ASTS S to which $[S]$ corresponds, $\mathcal{A}_\Phi$ can be seen as a predicate A_Φ over variables X, I_c and I_{uc}, and constitutes a solution to the discrete controller synthesis problem. Further, still according to Marchand and Samaan (2000), A_Φ is also the *maximally permissive controller w.r.t.* Φ, *i.e.,* the least constraint sufficient to avoid states in Bad.

Remark 3. From a computational point of view, to compute I_{Bad} (expressed here in an enumerated way), we need to be able to eliminate quantifiers ($\forall$, $\exists$) over variables, to intersect and complement predicates, as well as to compute the pre-image operator based on T. All these operations are easy to perform when dealing with Boolean variables only, but become intricate or even impossible when dealing with numerical variables. We shall also need to find a way to force the termination of the fix-point computation necessary to compute I_{Bad}.

2.3 Control of an ASTS: Over-approximation Solution

As seen in the previous section, the actual computation of the controller, which is based on a fix-point equation to compute I_{Bad}, is generally not possible for undecidability (or complexity) reasons. We use abstract interpretation techniques (see *e.g.,* (Cousot and Cousot, 1977); (Halbwachs et al., 1997); (Jeannet, 2003)) to overcome the undecidability problem, and compute an over-approximation I'_{Bad} of the fix-point I_{Bad}. This over-approximation ensures that the forbidden states Bad are not reachable in the controlled system. Thus, the synthesized controller remains correct, yet may not be maximally permissive *w.r.t.* the invariant. We illustrate this principle in Figure 2.

We shall now briefly present the abstract interpretation techniques and show how one can apply them to effectively compute a symbolic controller.

Overview of Abstract Interpretation Techniques Abstract interpretation techniques rely on Galois connections to approximate the solution of fix-points of the form $\mathrm{fp}(\lambda x.f(x))$, for a monotonic function f.

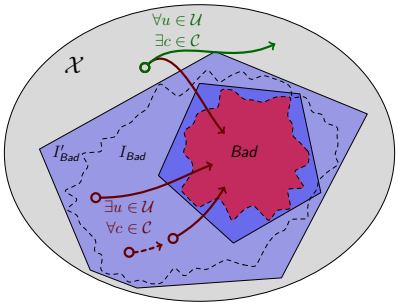

Fig. 2. Over-approximation I'_{Bad} of the possibly non-computable set of forbidden states I_{Bad}.

Definition 4. (Galois connection). *Let* $\langle D, \subseteq \rangle$ *and* $\langle \Lambda, \sqsubseteq \rangle$ *be partial orders, and let* $\alpha \colon D \to \Lambda$ *and* $\gamma \colon \Lambda \to D$ *be functions. Then* $D \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \Lambda$ *is a* Galois connection *iff*

$$\forall d \in D, \forall \ell \in \Lambda, d \subseteq \gamma(\ell) \Leftrightarrow \alpha(d) \sqsubseteq \ell$$

In abstract interpretation settings, D is the (possibly infinite) *concrete domain*, and exactly represents sets of states; Λ is the (possibly infinite) *abstract domain*, over-approximating these sets with a finite representation. $\sqcup$ (resp. $\sqcap$) denotes the *join* (resp. *meet*) operation, over-approximating in the abstract domain the union (resp. intersection) of sets. Note that $\langle \Lambda, \sqsubseteq \rangle$ associated with the above operations forms a lattice of which we denote by $\top$ the top and $\bot$ the bottom elements.

As the goal is to compute an over-approximation of (co-)reachable state space(s), we denote by $f^\sharp \colon \Lambda \to \Lambda$ the function "evaluating" a program statement $f \colon D \to D$ on abstract values (*i.e.,* such that $\forall \ell \in \Lambda, f^\sharp(\ell) \sqsupseteq \alpha \circ f \circ \gamma(\ell)$).

Also, given an abstract value $\ell \in \Lambda$ and a function $f^\sharp \colon \Lambda \to \Lambda$ such that $\forall \ell \in \Lambda$, $\ell \sqsubseteq f^\sharp(\ell)$, the *widening operator* $\nabla \colon \Lambda \times \Lambda \to \Lambda$ is such that $\mathrm{lfp}(\lambda l.\ell \sqcup f^\sharp(l)) \sqsubseteq \mathrm{lfp}(\lambda l.\ell \nabla f^\sharp(l))$.This operator guarantees the convergence of the fix-point computation in a finite number of steps.

In the remainder of this section we assume given an abstract domain Λ and associated operators capable of abstracting the state space of ITSs. We give further details about logico-numerical abstract domains in Section 3.1 bellow.

Computing $I'_{Bad} \supseteq I_{Bad}$ We extend the previous definitions to over-approximate the set of reachable states that may uncontrollably lead to a state violating the desired invariant.

In the abstract domain, given $\mathcal{T}_S^{-1\sharp}$ the abstract version of the pre-image function $\mathcal{T}_S^{-1}$ for the ITS $[S]$, we define $\mathrm{pre}_u^\sharp \colon \Lambda \to \Lambda$ as

$$\mathrm{pre}_u^\sharp(B) \stackrel{\text{def}}{=} \exists_{\mathcal{U}}^\sharp \left(\forall_{\mathcal{C}}^\sharp \left(\mathcal{T}_S^{-1\sharp}(B) \sqcap \alpha(\mathcal{A}_S) \right) \right)$$

where $\exists_{\mathcal{U}}^\sharp \ell$ (resp. $\forall_{\mathcal{C}}^\sharp \ell$) denotes the existential (resp. universal) quantification of every variables belonging to $\mathcal{U}$ (resp. $\mathcal{C}$), in the abstract value $\ell \in \Lambda$.

In the end, we have $I'_{Bad} = \gamma \circ \mathrm{coreach}_u^\sharp \circ \alpha(Bad)$, $\mathrm{coreach}_u^\sharp \colon \Lambda \to \Lambda$ being defined as:

$$\mathrm{coreach}_u^\sharp(B) \stackrel{\text{def}}{=} \mathrm{lfp}(\lambda \beta.B \sqcup \mathrm{pre}_u^\sharp(\beta))$$

As $\mathrm{coreach}_u^\sharp$ may not converge in finite time, we actually use the following over-approximating function to get I'_{Bad}:

$$\mathrm{coreach}_u^\nabla(B) \stackrel{\text{def}}{=} \mathrm{lfp}(\lambda \beta.B \nabla \mathrm{pre}_u^\sharp(\beta))$$

From I'_{Bad}, one can compute a controller A'_Φ, using the same technique as in the Boolean case.

3. FURTHER TECHNICAL CHOICES

3.1 Over-approximating Logico-numerical State Spaces

In the previous section, we assumed given an abstract domain capable of over-approximating the state spaces of ITSs. Let us now present actual abstract domains, first for state spaces exclusively consisting of numerical components, then when this space is also defined on Boolean variables.

Numerical Abstract Domains Several abstract domains have been proposed to over-approximate state spaces only defined on numerical variables. Common examples are *boxes* (Cousot and Cousot, 1976) and *convex polyhedra* (Cousot and Halbwachs, 1978).

The former kind of domains associates an interval to each numerical component of the state space; *i.e.,* it represents sets of states defined on the n-tuple of variables $\langle v_1, \ldots, v_n \rangle$ by a conjunction of constraints in the form of $\bigwedge_{i \in [1,n]} (a_i \leqslant v_i \leqslant b_i)$. For this domain, the *concretization function* (γ) is the identity, and the *abstraction function* (α) computes the smallest intervals containing a given concrete value. An abstract value equals $\top$ iff it is equivalent to $\bigwedge_{i \in [1,n]} (-\infty \leqslant v_i \leqslant \infty)$.

Convex polyhedra additionally capture relational linear constraints in the form of $\sum_{i \in [1,n]} a_i v_i \leqslant b$. For this domain, γ is also the identity function, whereas α computes the least convex polyhedron containing a given concrete value if it exists, or $\top$ (no constraint at all) otherwise.

Remark 4. Observe that the over-approximating algorithm presented above could be adapted to work on transition systems resulting from an enumeration of the Boolean state variables, i.e., with distinct locations. Purely numerical abstract domains such as the ones presented in this section would suffice to perform such synthesis. However, as this method would suffer from the state explosion problem, we use logico-numerical abstract domains *to avoid performing this enumeration:*

Logico-numerical Abstract Domains In order to handle logico-numerical state spaces without relying on an enumeration of the Boolean components, one has to compose abstract domains handling variables defined on different domains (Cousot and Cousot, 1979). Schrammel (2013) proposes two of such compositions to actually abstract concrete domains comprising both Boolean ($\mathbb{B}^n$) and numerical ($\mathcal{V}^m$) components:

A *product domain* $B \times A$ represents conjunctive relations between A and B. For the logico-numerical case, we would exploit the Galois connection $\wp(\mathbb{B}^n \times \mathcal{V}^m) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \wp(\mathbb{B}^n) \times \mathcal{N}$, with any numerical abstract domain $\mathcal{N}$.

Alternatively, a *power domain* B^A ($= A \to B$) associates one abstract value of B for each value of the set A. Operations on this domain are computationally more expensive than on the product; abstract values are also

more voluminous. In our case, we use $\wp(\mathbb{B}^n \times \mathcal{V}^m) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{B}^n \to \mathcal{N}$; with this representation, the Boolean components of the state space are represented exactly, whilst the numerical components are abstracted by a numerical abstract domain $\mathcal{N}$.

In practice, product domains can be implemented with a combination of *binary decision diagrams* and classical numerical abstract domains. Also, as *multi-terminal binary decision diagrams* (Clarke et al., 1993) can be used to describe functions in the form of $\mathbb{B}^n \to \mathcal{D}$ (for any $\mathcal{D}$ whose elements can be represented in a canonical way), this kind of structure is suitable to the implementation of power abstract domains.

3.2 Producing Actual Programs

As our goal is to obtain *executable* controllers, we eventually need to compute a new transition function so that the resulting system always evolves in I^c_{Bad}, and is deterministic. In terms of the ITS $[S]$, the input space of the resulting ITS is restricted to $\mathcal{U}$ (instead of $\mathcal{U} \times \mathcal{C}$ for $[S]$). Also, the new transition function $\mathcal{T}_\Phi \colon \mathcal{X} \times \mathcal{U} \to \mathcal{X}$ is obtained by combining the original one $\mathcal{T}_S$ and the constraints $\mathcal{A}_\Phi$ in the following way:

$$\mathcal{T}_\Phi(x, u) = \mathcal{T}_S(x, u, \mathit{choice} \circ \mathcal{K}_\Phi(x, u))$$

with $\mathcal{K}_\Phi(x, u) = \{c \mid (x, u, c) \in \mathcal{A}_\Phi\}$.

Note that, given a state x and an input u, the set computed by $\mathcal{K}_\Phi(x, u)$ may not be a singleton (*i.e.,* several controllable transitions from x may lead to states belonging to I^c_{Bad}). Here, the *choice*$\colon \wp(\mathcal{C}) \to \mathcal{C}$ function points out the need for specifying an additional policy for selecting between all possible options provided by the controller.

This choice can be done based on criteria depending on the actual application domain. For instance, randomly drawing an element from the set of valid controllable inputs works, but may also be unfair; quantitative objectives may also be taken into account to perform this choice, *e.g.,* for optimization purposes.

3.3 Restricting Controllable Inputs to Booleans

Our goal is to synthesize executable controllers; also, to integrate more easily in the BZR tool-suit (Delaval et al., 2013), it would be convenient to be able to produce actual programs not relying on a run-time constraints solver interpreting the controller. Besides, and on an even more practical level, existing libraries implementing operations on numerical abstract domains do not provide commodious means to universally quantify variables on abstract values.

So in practice, we restrict ourself to handle only Boolean controllable inputs ($\mathcal{C} = \mathbb{B}^n$). With this additional constraint on the system to control, we can restate our pre-image operator so that the universal quantification on $\mathcal{C}$ is performed in the concrete domain:

$$\mathrm{pre}^{\sharp'}_u(B) \stackrel{\text{def}}{=} \exists^\sharp_{\mathcal{U}} \alpha \left\{ (x, u) \,\middle|\, \forall c \in \mathcal{C}, (x, u, c) \in \gamma \left(\mathcal{T}_S^{-1\sharp}(B) \sqcap \alpha(\mathcal{A}_S) \right) \right\}$$

In addition, this restriction allows to *triangularize* the controller in order to statically generate code computing the *choice* $\circ\, \mathcal{K}_\Phi$ function. In that case, the triangularization method is similar to the one defined by Delaval et al. (2010).

Remark 5. Note that if the uncontrollable variables are Boolean variables as well and if only one of the input variables can be true at each instant, then we obtain a finite set of actions some of them being controllable and the other ones uncontrollable, as in the classical Ramadge & Wonham framework. In that case, and if applied to a system with distinct locations as explained in Remark 4, then our methodology is reduced to the one of Kalyon et al. (2011) and implemented in Smacs[3] *when all state variables are fully observable.*

3.4 On the Convexity of Bad

As we saw in the previous section, the computation of I_{Bad} strongly relies on a representation of the numerical constraints by means of convex polyhedra. However, starting from the predicate Φ, we need to compute the set of states Bad. Sometimes, this set of states can not be exactly represented in the abstract domain, leading to an over-approximation $\alpha(Bad)$ of Bad that might be too rough (*e.g.,* $\alpha(\{x \in \mathbb{Z} \mid x \leqslant 0 \wedge 10 \leqslant x\}) = \top$ with usual numerical abstract domains). To alleviate this problem, and to compute a more precise over-approximation of Bad, one can split Bad into a disjunction of n convex clauses Bad_i, compute every set I'_{Bad_i} independently as explained above, and then obtain the controller avoiding the states $\bigcup_{i \in [1,n]} Bad_i$ from $\bigcup_{i \in [1,n]} I'_{Bad_i}$.

4. IMPLEMENTATION

ReaX: Reactive System Control Synthesizer We have implemented the synthesis algorithms described above by extending an already existing tool, called ReaVer[4]. ReaVer is dedicated to the analysis of logico-numerical data-flow programs by using abstract interpretation techniques (Schrammel, 2013). It constitutes a framework to manipulate symbolic representations of logico-numerical formulas and functions, as well as abstract values.

Internally, ReaVer makes use of the CUDD library[5], involving *typed decision graphs* (Billon, 1987) for efficiently representing binary decision diagrams. It also uses the APRON library[6] (Jeannet and Miné, 2009), that features several numerical abstract domains such as boxes and convex polyhedra. Additionally, ReaVer uses the BddApron[7] library, allowing the representation of Boolean expressions with arithmetic constraints, and the combination of Boolean formulas (as binary decision diagrams) with numerical abstract domains in the form of products ($\wp(\mathbb{B}^n) \times \mathcal{N}$) and power ($\mathbb{B}^n \to \mathcal{N}$) domains (where $\mathcal{N}$ can be chosen among the domains provided by APRON).

ReaX reuses the internal structures of ReaVer to synthesize controllers by using either the Boolean or the over-approximating algorithms described in Section 2. In the Boolean case, it is also able to ensure other classical

[3] http://www.smacs.be/
[4] http://www.cs.ox.ac.uk/people/peter.schrammel/reaver/
[5] http://vlsi.colorado.edu/~fabio/CUDD/
[6] http://apron.cri.ensmp.fr/library/
[7] http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/

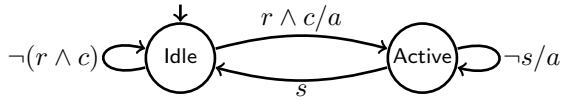

Fig. 3. Task automaton: s and r are non-controllable inputs, c is a controllable one, and the output a holds whenever the automaton enters or remains in the Active state. The controller can only prevent Task to enter in state Active, by setting $c = \mathit{false}$.

control objectives such as reachability and/or attractivity properties. In the over-approximating case, and as stated in Section 3.3, the implementation is restricted to programs with Boolean controllable inputs only. At last, ReaX is able to triangularize the controller in order to obtain a function computing the controllable inputs from the non-controllable ones and the current state (as detailed in Section 3.2).

BZR back-end We also implemented a new back-end to the BZR compiler [8], allowing to translate reactive programs with contracts into the input format of ReaX.

5. EVALUATIONS

In order to evaluate ReaX, we first present comparisons of execution times with SIGALI for the Boolean case [9]. Then, we detail a series of experiments on systems involving numerical variables. Eventually, we estimate the performance of ReaX on a realistic use case based on a model for the coordination of reactive self-adaptive systems.

5.1 Performance Comparison for the Boolean Case

Apart from allowing discrete controller synthesis for infinite-state systems, ReaX also performs very well in the finite case. We evaluate this claim before presenting an evaluation for the former case, by comparing the execution times of ReaX with the ones of SIGALI for the same set of input systems. Note that this comparison mostly emphasizes the efficiency of the CUDD library over the decision diagram package used within SIGALI.

To perform this evaluation, we use the following example: considering a set of n tasks modeled by n parallel instances of the Task automaton described in Figure 3, the property to enforce is the mutual exclusion between all Active states (*i.e.,* at most one output a can hold at a time). In this example, the number of tasks exactly represents the number of Boolean state variables.

We programmed this model in BZR for several values of n, and used them to generate the same synthesis problems suitable either to SIGALI or ReaX. We show in Figure 4 the time it takes for both tools to synthesize and triangularize a controller for various values of n. First, ReaX is globally more efficient than SIGALI on these benchmarks; also, the computation time of ReaX is more regular than SIGALI when the number of Boolean state variables grows.

These results show that, at least when the input system presents some regularity, the computation time of ReaX

[8] `http://bzr.inria.fr/`

[9] All timing measurements were carried out on a high-end desktop computer with 3.2GHz CPU cores and 6Go RAM — the synthesis algorithms of ReaX and SIGALI are single-threaded.

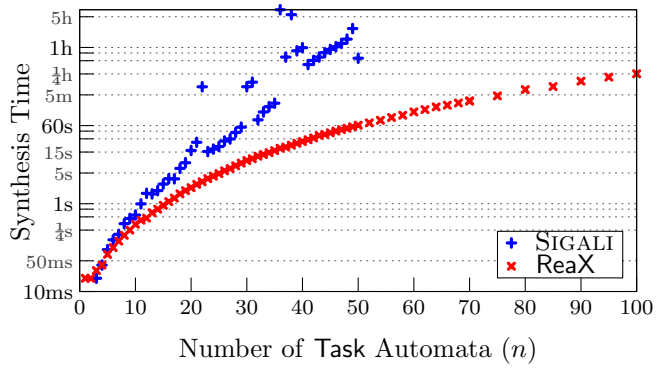

Fig. 4. Synthesis times for the parallel tasks example. (SIGALI takes more than 56h if $n = 51$.)

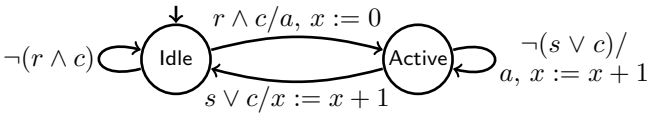

Fig. 5. Task' automaton: Task automaton of Figure 3, extended with a numerical variable x counting the number of steps since Active was entered. The controller can prevent Task' to enter in Active, and force it to enter in Idle.

is more predictable. (Of course, the theoretical worst case complexity of the invariance under control synthesis algorithm used by both tools is exponential in the number of variables, and there might be examples where SIGALI performs better than ReaX.)

5.2 Further Experiments for the Over-approximating Case

Setup In order to evaluate ReaX on systems involving numerical variables, we extend the previous tasks example by introducing one integer variable per Task automaton; it is represented in Figure 5. In the sequel, we denote by x_i the counter of the ith instance of Task' automaton.

We perform two sets of experiments based on this example, by restricting the invariance property in two steps:

(1) In addition to the mutual exclusion between all Active states, the invariance property is completed to restrict the value of each counter individually (meaning that a task must not be active for too long). This is achieved by enforcing the following conjunction of linear constraints: $\bigwedge_{i \in [1,n]} x_i \leqslant 10$;

(2) For the second sets of experiments, we additionally enforce relational constraints on the counters to impose interleaving restrictions (*e.g.,* imposing $x_1 \leqslant x_2$ means that x_1 must be reset at least once between two resets of x_2). For our example, we arbitrarily choose to add all constraints $x_i \leqslant x_{i+1}$, $\forall i \in [1, n[$.

On these examples, all synthesis we carried out with product domains failed due to too coarse approximations of the state space: too many intrinsic links between Boolean and numerical variables are lost when computing either $\alpha(\mathit{Bad})$ or I'_{Bad}. So, for each sets of experiments, ReaX is configured to use power domains with either boxes (intervals) or convex polyhedra numerical abstract domains.

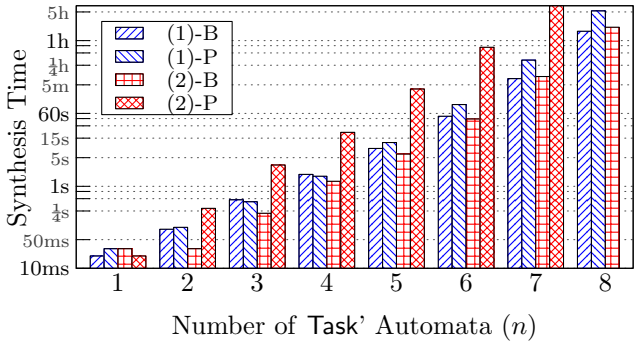

Fig. 6. Over-approximating synthesis times for the Task' experiments. In the key, "(e)-$\mathcal{N}$" means "experiment (e), with numerical domain $\mathcal{N}$" (B = boxes, P = convex polyhedra). Notice the logarithmic scale.

Synthesis Results Let us detail the results in the cases where $n = 2$. For the sake of clarity, we denote by t_1 ($\in \{\mathsf{Idle}, \mathsf{Active}\}$) the implicit Boolean state variable of the first instance of Task', and similarly for t_2. Also, we represent sets of states as predicates over these variables.

For experiments (1), the synthesis succeeds with boxes as well as with convex polyhedra; this result is as expected since the invariance property does not involve relational constraints (nor do the guards and numerical expressions in the automaton). In both cases, the allowed states are

$$I'^{c}_{Bad} = \left\{ \begin{array}{l} (t_1 = \mathsf{Idle} \wedge t_2 = \mathsf{Idle} \quad \wedge x_1 \leqslant 10 \wedge x_2 \leqslant 10) \vee \\ (t_1 = \mathsf{Idle} \wedge t_2 = \mathsf{Active} \wedge x_1 \leqslant 10 \wedge x_2 \leqslant \ \ 9) \vee \\ (t_1 = \mathsf{Active} \wedge t_2 = \mathsf{Idle} \wedge x_1 \leqslant \ \ 9 \wedge x_2 \leqslant 10) \end{array} \right\}$$

thus forbidding each Task' to stay in Active if its counter reaches 9 (this could otherwise lead in one transition to a state where $t_1 = \mathsf{Active} \wedge x_1 = 10$, then uncontrollably to a state where $x_1 = 11$ — and similarly for t_2 and x_2).

For experiments (2) (with relational constraints), the synthesis with boxes fails as it computes:

$$\begin{array}{l} I'_{Bad} \supseteq \{t_1 = \mathsf{Idle} \wedge t_2 = \mathsf{Idle} \wedge x_1 < 11 \wedge x_2 < 10\} \\ \quad\ \supseteq \{t_1 = \mathsf{Idle} \wedge t_2 = \mathsf{Idle} \wedge x_1 = \ \ 0 \wedge x_2 = \ \ 0\} = \mathcal{X}_0 \end{array}$$

On the contrary, the convex polyhedra abstract domain allows to capture the relational constraints, and the allowed states are

$$I'^{c}_{Bad} = \left\{ \begin{array}{l} (t_1 = \mathsf{Idle} \wedge t_2 = \mathsf{Idle} \quad \wedge x_1 \leqslant x_2 \leqslant 10) \vee \\ (t_1 = \mathsf{Idle} \wedge t_2 = \mathsf{Active} \wedge x_1 \leqslant x_2 \leqslant \ \ 9) \vee \\ (t_1 = \mathsf{Active} \wedge t_2 = \mathsf{Idle} \wedge x_1 < x_2 \leqslant 10) \end{array} \right\}$$

hence reflecting the linear constraints involving x_1 and x_2.

Note that incidentally, all controllers successfully computed during these experiments are maximally permissive.

Performance Results We report in Figure 6 the whole execution times (synthesis, plus triangulation of the controller in case of success) for each series of experiments, and various numbers of parallel Task' automata. Notice the number of such automata reflects exactly the number of both Boolean and numerical state variables in the ASTS to control. As expected, convex polyhedra induce longer synthesis times than boxes since they involve more intricate computations. The total memory occupation is also significantly higher when using convex polyhedra instead of boxes (1.2GB *vs.* 160MB for $n = 7$ in experiments (2)).

	Number of Boolean State Variables	SIGALI	ReaX
2-tiers	39	6s	3.4s
4-tiers	73	113s	14s

Table 1. Synthesis times for n-tier autonomic management systems.

5.3 Realistic Reactive Adaptation Use Case

In order to start investigating towards the application of the innovative capabilities of ReaX to a realistic use case, we present in this section some preliminary evaluations in the context of self-adaptive distributed computing systems. Indeed, the manual administration of such systems tends to become more and more complex, and it is thus interesting to make computing systems do this difficult task; this is the role of *autonomic management systems*. Such systems are intended to detect events through measurements on a computing system to manage (*e.g.,* underloads, overloads, failures), take management decisions, and then act upon it (*e.g.,* by allocating, reclaiming, or repairing resources).

However, in these software systems, management decisions are usually taken by components designed independently from each other. As a consequence, they take decisions based on a limited knowledge about the managed system, possibly leading to situations where two inconsistent or counter-productive actions are simultaneously performed. For instance, a component can decide to reclaim a resource because it detects that it is not necessary for the correct behavior of the managed system, even though its underuse is imputable to a failure being actually repaired by another component of the management system.

Now, reactive programming techniques suit the needs for the design of such self-adaptive software systems, and a new methodology based on these techniques has been proposed for designing coordinated autonomic management systems in the context of the French ANR project Ctrl-Green [10]. It makes use of synchronous languages to model the managed system as well as express management decisions, and ensures the correct coordination of the latter by means of discrete controller synthesis. The proposed design methodology has successfully been applied to automate the management of realistic distributed multi-tier applications (*e.g.,* a 4-tiers computing system consisting of several chained services such as: a server handling static web pages and sending requests to replicated application servers rendering dynamic contents, themselves consulting database servers through a proxy). As the proposed models are finite and available as BZR programs, we can exercise ReaX on them, and make the most of the opportunity to compare synthesis times with the ones of SIGALI.

Preliminary Evaluation Results We present in Table 1 the synthesis times for reactive programs aimed at managing 2 and 4-tiers systems. By comparing these synthesis times with the ones of Figure 4 for the same amount of Boolean state variables ($\approx n$), we can observe that the two realistic models entail less intricate behaviors and constraints than our tasks example. Hence, even more complicated programs might be within reach of ReaX.

[10] `http://www.ctrlgreen.org/`

6. CONCLUSIONS AND FURTHER WORKS

In this paper, we proposed symbolic algorithms for the synthesis of controllers ensuring safety properties of infinite state systems modeled by ASTSs. In order to overcome the undecidability issue, we rely on abstract interpretation techniques and compute an over-approximation of the forbidden states. We use logico-numerical abstract domains to handle ASTSs with Boolean and numerical state variables while avoiding the state explosion problem induced by enumerating the Boolean state space. Our tool ReaX implements these techniques and allows to enforce properties on systems handling data. Evaluations indicate that it can favorably replace Sigali. We also present promising preliminary evaluations towards the use of ReaX in the context of self-adaptive systems.

Further works can be organized in two categories. First, on an algorithmic level, we plan to explore the use of abstract acceleration techniques (Schrammel and Jeannet, 2012) to overcome the loss of precision induced by the widening operations. We also want to adapt our algorithms to be able to synthesize controllers ensuring deadlock-freeness for ASTSs. Secondly, we plan to extend the reactive adaptation use case to derive management strategies and coordination policies from optimization objectives or constraints expressed on quantitative measures. This aspect of the work would entail extending ReaX to handle optimal control objectives.

REFERENCES

Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. (2003). The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 91(1), 64–83.

Billon, J. (1987). Perfect normal forms for discrete programs. Technical report, Bull.

Cassandras, C. and Lafortune, S. (2007). *Introduction to Discrete Event Systems*. Springer.

Cassez, F., David, A., Fleury, E., Larsen, K., and Lime, D. (2005). Efficient on-the-fly algorithms for the analysis of timed games. In *Conf. on Concurrency Theory (CONCUR)*, volume 3653 of *LNCS*, 66–80.

Clarke, E.M., McMillan, K.L., Zhao, X., Fujita, M., and Yang, J. (1993). Spectral transforms for large boolean functions with applications to technology mapping. In *30th Conference on Design Automation*, 54–60. IEEE.

Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, 106–130. Dunod, Paris, France.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, POPL '77, 238–252.

Cousot, P. and Cousot, R. (1979). Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, 269–282.

Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, POPL '78, 84–96.

Delaval, G., Marchand, H., and Rutten, E. (2010). Contracts for modular discrete controller synthesis. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, 57–66.

Delaval, G., Rutten, E., and Marchand, H. (2013). Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, 23(4), 385–418.

Halbwachs, N., Proy, Y.E., and Roumanoff, P. (1997). Verification of Real-Time Systems using Linear Relation Analysis. *Form. Methods Syst. Des.*, 11, 157–185.

Holloway, L., Krogh, B., and Giua, A. (1997). A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems: Theory and Application*, 7, 151–190.

Jeannet, B. (2003). Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Reactive Systems. *Form. Methods Syst. Des.*, 23, 5–37.

Jeannet, B. and Miné, A. (2009). Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, 661–667. Springer.

Kalyon, G., Le Gall, T., Marchand, H., and Massart, T. (2011). Symbolic supervisory control of infinite transition systems under partial observation using abstract interpretation. *Discrete Event Dynamic Systems : Theory and Applications*, 22(2), 121–161.

Kumar, R., Garg, V., and Marcus, S. (1993). Predicates and predicate transformers for supervisory control of discrete event dynamical systems. *IEEE Trans. Autom. Control*, 38(2), 232–247.

Kumar, R. and Garg, V. (2005). On computation of state avoidance control for infinite state systems in assignment program model. *IEEE Trans. on Automation Science and Engineering*, 2(2), 87–91.

Le Gall, T., Jeannet, B., and Marchand, H. (2005). Supervisory control of infinite symbolic systems using abstract interpretation. In *Proceedings of the 44th IEEE Conference on Decision and Control*, 31–35.

Li, Y. and Wohnam, W. (1994). Control of vector discrete-event systems-part ii : controller synthesis. *IEEE Trans. Automatic Control*, 39(3), 512–531.

Marchand, H., Bournai, P., Le Borgne, M., and Le Guernic, P. (2000). Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4), 325–346.

Marchand, H. and Samaan, M. (2000). Incremental Design of a Power Transformer Station Controller Using a Controller Synthesis Methodology. *IEEE Trans. Softw. Eng.*, 26, 729–741.

Miremadi, S., Akesson, K., and Lennartson, B. (2008). Extraction and representation of a supervisor using guards in extended finite automata. In *9th International Workshop on Discrete Event Systems*, 193–199.

Ramadge, P.J. and Wonham, W.M. (1989). The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77(1), 81–98.

Schrammel, P. (2013). *Logico-Numerical Verification Methods for Discrete and Hybrid Systems*. Ph.D. thesis, University of Grenoble.

Schrammel, P. and Jeannet, B. (2012). Applying Abstract Acceleration to (Co-)Reachability Analysis of Reactive Programs. *J. Symb. Comput.*, 47(12), 1512–1532.