Most General Property-Preserving Updates

Davide Bresolin, Ivan Lanese

Most General Property-Preserving Updates*

Davide Bresolin[1] and Ivan Lanese[2]

[1] University of Padova, Italy
Email: davide.bresolin@unipd.it
[2] Focus Team, University of Bologna/INRIA, Italy
Email: ivan.lanese@gmail.com

Abstract. Systems need to be updated to last for a long time in a dynamic environment, and to cope with changing requirements. It is important for updates to preserve the desirable properties of the system under update, while possibly enforcing new ones.
Here we consider a simple yet general update mechanism, which replaces a component of the system with a new one. The context, i.e., the rest of the system, remains unchanged. We define contexts and components as Constraint Automata interacting via either asynchronous or synchronous communication, and we express properties using Constraint Automata too. Then we build most general updates which preserve specific properties, considering both a single property and all the properties satisfied by the original system, in a given context or in all possible contexts.

1 Introduction

Update is a relevant topic [19], both for automatic updates, as in the context of adaptive systems [17] or autonomic computing [15], and for manual updates. A main reason is that one wants systems to last for a long time in a changing environment and to satisfy changing user requirements. However, a main point, namely correctness of the system after update, has received scarce attention till now, as remarked also in [14].

In this paper we consider a very simple yet general update mechanism, which replaces a part of the system with a new one. Formally, the system is seen as a context $\mathcal{C}$ containing the component to be updated $\mathcal{A}$, i.e., the system has the form $\mathcal{C}[\mathcal{A}]$. An update replaces $\mathcal{A}$ with $\mathcal{B}$, thus the system upon update has the shape $\mathcal{C}[\mathcal{B}]$. A basic question is: how to build a most general $\mathcal{B}$ such that if $\mathcal{C}[\mathcal{A}]$ satisfies a given property Φ, then also $\mathcal{C}[\mathcal{B}]$ satisfies the same property? This question is answered in Section 3. Note that $\mathcal{C}[\mathcal{B}]$ may satisfy further properties that $\mathcal{C}[\mathcal{A}]$ does not satisfy. The answer to the question above, which relates $\mathcal{A}$ and $\mathcal{B}$, depends both on the context $\mathcal{C}$ and on the property Φ. From this observation two generalizations emerge naturally. On one side, one may ask how to build a most general $\mathcal{B}$ such that for a given context $\mathcal{C}$, *all the properties* satisfied by $\mathcal{C}[\mathcal{A}]$ are also satisfied by $\mathcal{C}[\mathcal{B}]$ (Section 3). We call such an update correct for a

* The authors acknowledge the support from the Italian INdAM – GNCS project 2016 *Logic, automata and games for self-adaptive systems.*

given context w.r.t. any property. On the other side, one may ask how to build a most general $\mathcal{B}$ such that *for each context* $\mathcal{C}$ if $\mathcal{C}[\mathcal{A}]$ satisfies property Φ, then $\mathcal{C}[\mathcal{B}]$ satisfies the same property (Section 4). We say that this is a correct update (w.r.t. the property Φ) that can be applied in any context. Finally, one may combine the two generalizations asking how to build a most general $\mathcal{B}$ to ensure correctness of update *in any context and w.r.t. any property* (Section 4).

The questions above are very general, and the detailed answer depends on the choice of the model for components and contexts, of the composition operators, and of the formalism for expressing properties. We consider here components, contexts and properties represented as Constraint Automata [6, 5], which have been used in the literature, e.g., to give a formal semantics to REO connectors [3] and Rebeca actors [20]. We consider both asynchronous and synchronous composition for components and contexts. We leave the systematic exploration of the research space above to future work. We illustrate the results of our approach by means of a simple running example. All the operations on Constraint Automata were computed using the tool GOAL [21], an interactive tool for defining and manipulating automata, which we extended to deal with Constraint Automata. Technical details not included in the paper for space reasons can be found in [9].

2 Constraint Automata

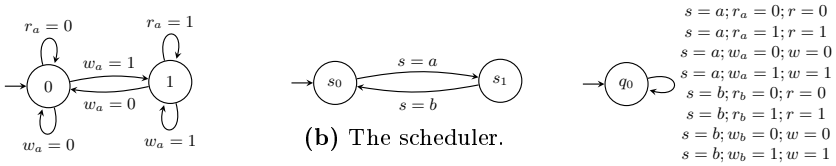
We model components, contexts and properties as Constraint Automata (CAs) [6], defined below. Throughout the paper we assume a finite set Data of *data values* which can be communicated and a finite set of states for the CAs. As in [6], the finiteness assumption is needed for the effectiveness of our constructions.

Definition 1 (Constraint Automata).
A constraint automaton $\mathcal{A}$ is a tuple $\langle Q, N, q_0, \rightarrow \rangle$ where:

1. *Q is a finite set of* states*;*
2. *N is a finite set of* node names *representing the interface between the CA and the outside world;*
3. *$q_0 \in Q$ is the* initial state*;*
4. *$\rightarrow \subseteq Q \times \mathsf{CIO}(N) \times Q$ is the* transition relation*, where $\mathsf{CIO}(N)$ is the set of* concurrent I/O operations *$c : N \mapsto \mathsf{Data} \cup \{\bot\}$ mapping every node in N to an element of Data, or to $\bot$ if no data is written/read. We assume that c is never the constant function with value $\bot$.*

Transitions of a CA are of the form $q \xrightarrow{c} p$, where c is a concurrent I/O operation. A *run* of a CA is a finite/infinite sequence $\rho = q_0 \xrightarrow{c_0} q_1 \xrightarrow{c_1} \ldots$ such that q_0 is the initial state and, for every i, $q_i \xrightarrow{c_i} q_{i+1}$ is a transition of the CA. In this case, we say that ρ *accepts* the *trace* $w = c_0 c_1 \ldots$. The *language* of a CA $\mathcal{A}$, denoted $\mathscr{L}(\mathcal{A})$, is the set of traces accepted by $\mathcal{A}$. Since a prefix of a run is again a run, languages are closed under prefix.

Given $c \in \mathsf{CIO}(N)$, we define $\mathsf{Nodes}(c)$ as the set of nodes through which data flow, formally $\mathsf{Nodes}(c) = \{n \in N \mid c(n) \neq \bot\}$. The *domain restriction* of a concurrent I/O operation c on a subset of nodes $N' \subseteq N$ is written $c{\downarrow_{N'}}$.



(a) The register a.

(b) The scheduler.

(c) The synchronizer.

Fig. 1. The CAs for Example 1.

Given two disjoint sets of nodes N_1 and N_2, and two CIOs $c_1 \in \mathsf{CIO}(N_1)$ and $c_2 \in \mathsf{CIO}(N_2)$, we define their *union* as the unique CIO $c_1 \cup c_2 \in \mathsf{CIO}(N_1 \cup N_2)$ such that $(c_1 \cup c_2)\downarrow_{N_1} = c_1$ and $(c_1 \cup c_2)\downarrow_{N_2} = c_2$.

Example 1. We introduce here our running example.

We consider a system which allows one to read and write information from/to two one-bit registers, denoted as a and b. The system is the composition of four components represented in Figure 1: two registers (only register a is shown, b is analogous), a scheduler that determines which register is active, and a synchronizer that communicates with the registers and the scheduler and proposes to the outside world the nodes r (read) and w (write) to access the active register. Labels on edges represent CIOs, written as semicolon-separated sets of assignments. Each assignment $n = d$ specifies that the data value d is communicated on node n. No communication occurs on nodes that do not appear in the label.

Registers are two-state CAs communicating with the synchronizer on nodes r_a (read a) and w_a (write a) for register a, and on nodes r_b and w_b for register b. The scheduler interacts with the synchronizer on node s. Essentially, the two registers are scheduled in round-robin order a, b. The synchronizer is a one-state CA that forwards the external operations to the currently active register.

We use CAs also to describe properties, which are prefix-closed sets of (finite or infinite) traces. We represent a property as a CA Φ accepting the corresponding set of traces. We say that a CA $\mathcal{A}$ satisfies the property Φ, written $\mathcal{A} \models \Phi$, iff $\mathscr{L}(\mathcal{A}) \subseteq \mathscr{L}(\Phi)$. CAs are as expressive as the safety linear μ-calculus [16] and, as a consequence, more expressive of the safety fragment of temporal logics like LTL and CTL.

2.1 Composition of CAs

We consider here a particular type of composition, where a component is embedded in a context. We examine two forms of synchronization between the context and the component: *synchronous* and *asynchronous*. Formally, we assume to have two CAs: $\mathcal{A}$ (the component) and $\mathcal{C}$ (the context). We also assume two disjoint finite sets of node names U and O. Communication between the component and the context goes through U, while communication between the context and the external world goes through O.

In the asynchronous case, at every step the context communicates either with the component via nodes in U, or with the external world via nodes in O, or with both at the same time. In the synchronous case, at every step the context communicates with both the component and the external world.

The embedding of $\mathcal{A}$ in $\mathcal{C}$ is defined by means of two operations on CAs [5]: projection and (synchronous or asynchronous) join.

Definition 2 (Asynchronous Join). *The* asynchronous join *of two CAs* $\mathcal{A} = \langle Q_A, U, q_0^A, \rightarrow_A \rangle$ *and* $\mathcal{C} = \langle Q_C, U \cup O, q_0^C, \rightarrow_C \rangle$ *is defined as the CA* $\mathcal{A} \bowtie_a \mathcal{C} = \langle Q_A \times Q_C, U \cup O, (q_0^A, q_0^C), \rightarrow_a \rangle$ *such that:*

- $(q,p) \xrightarrow{c}_a (q',p')$ *if* $\mathsf{Nodes}(c) \cap U \neq \emptyset$, $q \xrightarrow{c \downarrow_U}_A q'$ *and* $p \xrightarrow{c}_C p'$;
- $(q,p) \xrightarrow{c}_a (q,p')$ *if* $\mathsf{Nodes}(c) \cap U = \emptyset$ *and* $p \xrightarrow{c}_C p'$.

Definition 3 (Synchronous Join). *The* synchronous join *of two CAs* $\mathcal{A} = \langle Q_A, U, q_0^A, \rightarrow_A \rangle$ *and* $\mathcal{C} = \langle Q_C, U \cup O, q_0^C, \rightarrow_C \rangle$ *is defined as the CA* $\mathcal{A} \bowtie_s \mathcal{C} = \langle Q_A \times Q_C, U \cup O, (q_0^A, q_0^C), \rightarrow_s \rangle$ *such that:*

- $(q,p) \xrightarrow{c}_s (q',p')$ *if* $\mathsf{Nodes}(c) \cap U \neq \emptyset$, $\mathsf{Nodes}(c) \cap O \neq \emptyset$, $q \xrightarrow{c \downarrow_U}_A q'$ *and* $p \xrightarrow{c}_C p'$.

Given a CA $\mathcal{B}$ with node names from a set $U \cup O$, the projection on O removes the nodes in U from the interface of $\mathcal{B}$ and hides the communications occurring at those nodes. To define the projection, we need the relation $\rightsquigarrow_O^* \subseteq Q \times Q$, which is the smallest relation such that:

- $q \rightsquigarrow_O^* q$ for each $q \in Q$;
- if $q \rightsquigarrow_O^* p$ and $p \xrightarrow{c} r$ with $\mathsf{Nodes}(c) \cap O = \emptyset$, then $q \rightsquigarrow_O^* r$.

Definition 4 (Projection). *The* projection *of a CA* $\mathcal{B} = \langle Q, U \cup O, q_0, \rightarrow \rangle$ *on* O *is defined as the CA* $\mathcal{B}{\downarrow_O} = \langle Q, O, q_0, \rightarrow^* \rangle$ *such that* $q \xrightarrow{c}{}^* p$ *iff there exists* $d \in \mathsf{CIO}(U \cup O)$, $r \in Q$ *such that* $d{\downarrow_O} = c$ *and* $q \rightsquigarrow_O^* r \xrightarrow{d} p$.

The *asynchronous embedding* $\mathcal{C}[\mathcal{A}]_a$ and the *synchronous embedding* $\mathcal{C}[\mathcal{A}]_s$ of the component $\mathcal{A} = \langle Q_A, U, q_0^A, \rightarrow_A \rangle$ in the context $\mathcal{C} = \langle Q_C, U \cup O, q_0^C, \rightarrow_C \rangle$ are defined as

$$\mathcal{C}[\mathcal{A}]_a = (\mathcal{A} \bowtie_a \mathcal{C}){\downarrow_O} \qquad \mathcal{C}[\mathcal{A}]_s = (\mathcal{A} \bowtie_s \mathcal{C}){\downarrow_O}$$

The above definitions hide all nodes of the component and expose only the nodes from O. We will drop the subscript a or s to refer to both kinds of embedding.

Example 2. We can now build the system outlined in Example 1 by embedding the scheduler into the context, which is obtained by embedding the two registers (in any order) into the synchronizer. All the embeddings are asynchronous.

The states of the whole system, represented in Figure 2, are tuples (s_i, v_a, v_b) where s_i is the state of the scheduler and v_a and v_b are the values of the registers a and b, respectively.

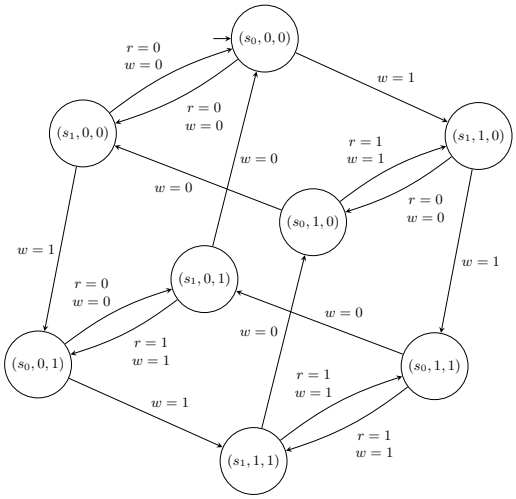


Fig. 2. Embedding of the scheduler in the context.

2.2 Determinization and complementation of CAs

In the next sections, we will need to complement CAs. Unfortunately, CAs are not closed under complementation. We solve the problem following the approach in [5], reported below.

Given a nondeterministic CA $\mathcal{A}$, by using the standard *subset construction* for finite word automata it is possible to obtain an equivalent deterministic CA $\mathsf{Subset}(\mathcal{A})$ that, in the worst case, is exponentially larger than $\mathcal{A}$.

We can complement $\mathsf{Subset}(\mathcal{A})$ by enriching it with a set of final states $F \subseteq Q$ and a *Büchi acceptance condition*. We say that a finite run is accepting whenever the last state of the run is final, while an infinite run is accepting if the set of final states F is visited infinitely often. Formally, given a deterministic CA $\mathcal{A} = \langle Q, N, q_0, \rightarrow_A \rangle$ we can build a CA with final states $\overline{\mathcal{A}} = \langle Q_\perp, N, q_0, \rightarrow_{\overline{A}}, F \rangle$ accepting the complement language as follows:

- $Q_\perp = Q \cup \{q_\perp\}$ where $q_\perp$ is a distinguished *sink state* not included in Q;
- $F = \{q_\perp\}$ (only the sink state is final);
- $q \xrightarrow{c}_{\overline{A}} q'$ iff $q \xrightarrow{c}_A q'$;
- $q \xrightarrow{c}_{\overline{A}} q_\perp$ for all $q \in Q$ and $c \in \mathsf{CIO}(N)$ such that there is no q' such that $q \xrightarrow{c}_A q'$;
- $q_\perp \xrightarrow{c}_{\overline{A}} q_\perp$ for all $c \in \mathsf{CIO}(N)$.

In the following we will need to compute expressions of the form $\mathcal{C}[\overline{\mathcal{A}}]$. This can be done by using the construction for standard CAs, and by choosing as final states of the result the set $Q_C \times \{q_\perp\}$, where Q_C is the set of states of $\mathcal{C}$ and $q_\perp$ is the sink state of $\overline{\mathcal{A}}$.[3]

[3] This construction is not correct for general CAs with final states, but it is correct in this restricted case [5].

We will also use the following operations on deterministic CAs with final states. $\mathsf{Prefix}(\mathcal{A})$ is the CA obtained by removing all non-final states from $\mathcal{A}$ and taking the connected component including the initial state. Notably, $\mathscr{L}(\mathsf{Prefix}(\mathcal{A}))$ is the maximal prefix-closed language included in $\mathscr{L}(\mathcal{A})$. $\mathsf{Switch}(\mathcal{A})$ is the CA with final states obtained from $\mathcal{A}$ by selecting as final states the non-final states of $\mathcal{A}$, and vice versa.

3 Updates Correct for a Given Context

Given a system $\mathcal{C}[\mathcal{A}]$, an update replacing $\mathcal{A}$ with $\mathcal{B}$ is *correct* w.r.t. a property Φ iff whenever $\mathcal{C}[\mathcal{A}] \models \Phi$ also $\mathcal{C}[\mathcal{B}] \models \Phi$. We assume that $\mathcal{A}$ and $\mathcal{B}$ have the same interface, that is, the same set of node names. This is not restrictive since one can always add node names that are never used.

This section considers both the cases "all properties, given context" and "given property, given context". We show that they can be both reduced to instances of the following problem: given a context $\mathcal{C}$ and a specification $\mathcal{S}$ representing the correct behavior of the whole system, find the $\mathcal{B}$s such that $\mathcal{C}[\mathcal{B}] \models \mathcal{S}$. By definition of $\models$, these are the solutions of the following language inequation:

$$\mathscr{L}(\mathcal{C}[\mathcal{B}]) \subseteq \mathscr{L}(\mathcal{S}) \tag{1}$$

Among all such $\mathcal{B}$s we select one generating the largest language, and we call it a *most general solution* of the inequation. Such a solution is unique up to language equivalence.

Lemma 1. *For each context $\mathcal{C}$ and specification $\mathcal{S}$, Inequation* (1) *has a unique most general solution, up to language equivalence.*

In Inequation (1), when $\mathcal{S}$ is the system before the update $\mathcal{C}[\mathcal{A}]$ we are in the setting "all properties, given context", while when $\mathcal{S}$ is a CA representing a given property Φ we are in the setting "given property, given context".

Inequation (1) has been studied by the logic synthesis and controller design communities, where it is known as the "unknown component problem" [22]. The following result is part of the theory developed in [22].

Theorem 1. *$\mathcal{B}$ is a solution of Inequation* (1) *iff* $\mathscr{L}(\mathcal{B}) \subseteq \overline{\mathscr{L}(\mathcal{C}[\overline{\mathcal{S}}])}$.

The literature does not provide, for our setting, a constructive way of building a most general CA satisfying the constraint above. We propose one below. One would expect that a most general CA is $\overline{\mathcal{C}[\overline{\mathcal{S}}]}$. However, since CAs are not closed under complementation, such a CA in general cannot be built. We show that $\mathsf{Prefix}(\mathsf{Switch}(\mathsf{Subset}(\mathcal{C}[\overline{\mathcal{S}}])))$ is the best possibile approximation which is a CA.

Theorem 2. *$\mathcal{B} = \mathsf{Prefix}(\mathsf{Switch}(\mathsf{Subset}(\mathcal{C}[\overline{\mathcal{S}}])))$ is a most general CA such that $\mathscr{L}(\mathcal{B}) \subseteq \overline{\mathscr{L}(\mathcal{C}[\overline{\mathcal{S}}])}$.*

Theorems 3 and 5 below show that $\mathcal{B}$ is a most general update correct for a given context $\mathcal{C}$. The former considers a given property Φ, the latter any property representable as a CA.

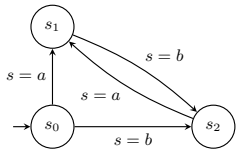


Fig. 3. Most general scheduler of Example 3.

Theorem 3. *Given a system* $\mathcal{C}[\mathcal{A}]_x$ *with* $x \in \{a, s\}$ *and a property* Φ *such that* $\mathcal{C}[\mathcal{A}]_x \models \Phi$, $\mathcal{B} = \mathsf{Prefix}(\mathsf{Switch}(\mathsf{Subset}(\mathcal{C}[\overline{\Phi}]_x)))$ *is a most general CA such that replacing* $\mathcal{A}$ *with* $\mathcal{B}$ *is a correct update w.r.t.* Φ.

Note that the above characterization does not depend on $\mathcal{A}$. However, if $\mathcal{C}[\mathcal{A}]$ does not satisfy the property Φ then every update is correct. Indeed the construction works for any property Φ, which may or may not hold for $\mathcal{C}[\mathcal{A}]$. Thus the approach can also be applied to ensure that new safety properties will hold after the update, e.g., to fix a bug or close a security vulnerability.

Theorem 4. *Given a system* $\mathcal{C}[\mathcal{A}]_x$ *with* $x \in \{a, s\}$ *and a property* Φ, $\mathcal{B} = \mathsf{Prefix}(\mathsf{Switch}(\mathsf{Subset}(\mathcal{C}[\overline{\Phi}]_x)))$ *is a most general CA such that replacing* $\mathcal{A}$ *with* $\mathcal{B}$ *ensures that* Φ *holds in* $\mathcal{C}[\mathcal{B}]_x$.

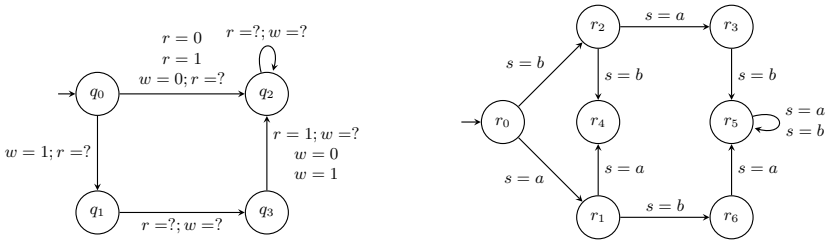
Theorem 5.
Given a system $\mathcal{C}[\mathcal{A}]_x$ *with* $x \in \{a, s\}$, $\mathcal{B} = \mathsf{Prefix}(\mathsf{Switch}(\mathsf{Subset}(\mathcal{C}[\overline{\mathcal{C}[\mathcal{A}]_x}]_x)))$ *is a most general CA such that replacing* $\mathcal{A}$ *with* $\mathcal{B}$ *is a correct update w.r.t. any property.*

Example 3. We can apply Theorem 5 to obtain a most general update for the case "given context, all properties" of the system in Example 1. By minimizing the result (up to language equivalence) we obtain the CA in Figure 3, where s_0 is the initial state. The solution recognizes the traces where one of the sequences $abababa\ldots$ and $bababab\ldots$ is communicated on node s. This implies that, e.g., replacing the original scheduler with a new one activating the registers in round-robin order b, a is a correct update. This matches the intuition, since the two registers are identical and swapping when they are accessible has no visible effect. Instead, using a scheduler that, e.g., always activates a and never activates b is not. A property falsified by this incorrect update is, for instance, P1 = "if *w=1* is executed at the first step, then at the third step *r=0* cannot be executed".

Example 4. Consider the property P1 above. It can be formalized by the CA Φ in Figure 4a. There, we use ? to denote 0, 1 or $\bot$, and we assume that at least one node in each constraint has non $\bot$ value. The system of Example 1 satisfies Φ. We want to characterize the updates that preserve Φ.

We can apply Theorem 3 to obtain the most general scheduler depicted in Figure 4b. Notice that it accepts the following computations:

- any computation of length at most 2: in this case the third step is never reached, and the property is trivially satisfied;



(a) CA Φ for P1. **(b)** Most general scheduler.

Fig. 4. CAs of Example 4.

– any computation that starts with $s = a, s = b, s = a$ or $s = b, s = a, s = b$: in this case the value 1 written in the register at the first step is not changed in the second step, and made available in the third step.

We now move to the study of the complexity of our construction.

Theorem 6. *Given a system $\mathcal{C}[\mathcal{A}]_x$ with $x \in \{a, s\}$ finding a most general $\mathcal{B}$ such that replacing $\mathcal{A}$ with $\mathcal{B}$ is a correct update for a given property Φ or for any property, or an update that makes Φ hold is in 2-EXPTIME.*

The 2-EXPTIME complexity arises from a double subset construction.

Theorem 7. *Given a system $\mathcal{C}[\mathcal{A}]_x$ with $x \in \{a, s\}$ and a property Φ such that $\mathcal{C}[\mathcal{A}]_x \models \Phi$, finding a most general $\mathcal{B}$ such that replacing $\mathcal{A}$ with $\mathcal{B}$ is a correct update w.r.t. Φ, or that makes Φ hold is EXPSPACE-hard.*

The lower bound is proved by reducing a suitable three-player game to Inequation (1). The game is played on a finite-state graph, with the first player (the component) and the third player (the specification) in a coalition against the second player (the context). At every round of the game, given the current state, the successor state is determined by the choice of moves of the players. A suitable safety condition establishes who wins the game. The reduction shows that a winning strategy for Player 1 corresponds to a correct update of the system, if $\mathcal{C}[\mathcal{A}]_x \models \Phi$, and to an update that makes Φ hold otherwise. The problem of finding a winning strategy in this game is EXPSPACE-complete [10]. The details of the reduction can be found in [9].

Theorem 7 deals with the case "given property, given context". It seems not easy to adapt the reduction to the case "all properties, given context". Finding a lower bound for the latter case is an open problem.

4 Updates Correct for all Contexts

In this section we study both the cases "given property, all contexts" and "all properties, all contexts". Similarly to the previous section, we can assume w.l.o.g.

that $\mathcal{A}$ and $\mathcal{B}$ have the same interface U, and that all the contexts we consider have U as internal interface.

Let us start from the case of a given property. The property defines a minimum set of node names O required for the external interface of the context. For some properties, replacing $\mathcal{A}$ with $\mathcal{B}$ is a correct update iff the traces of $\mathcal{B}$ are included in the traces of $\mathcal{A}$. However, this is not the case for all the properties. For instance, all the updates are correct w.r.t. the properties $\mathbf{tt} = \mathsf{CIO}(O)^* \cup \mathsf{CIO}(O)^\omega$ or $\mathbf{ff} = \emptyset$. Indeed, in the asynchronous case these are the only possibilities.

Theorem 8. *Let Φ be a property and $\mathcal{A}$ a CA. For the asynchronous embedding, the most general CA such that replacing $\mathcal{A}$ with $\mathcal{B}$ is a correct update w.r.t. Φ in all the contexts is* $\mathbf{tt}$ *if Φ is either* $\mathbf{tt}$ *or* $\mathbf{ff}$*, $\mathcal{A}$ otherwise.*

In the synchronous case the context and the component progress in lock-step. Given a property Φ, there are steps i in the computation on which Φ does not pose any restriction: if a trace z of length $i-1$ is in $\mathscr{L}(\Phi)$, then all the traces of length i having z as prefix are also in $\mathscr{L}(\Phi)$. Conversely, there are steps where Φ observes the system and whether a trace of length i is in $\mathscr{L}(\Phi)$ or not depends on the last action of the system. The *observation-point language* contains all the traces whose length identifies an observation point. Since the component $\mathcal{A}$ communicates on the internal interface U, the observation-point language is defined on the alphabet $\mathsf{CIO}(U)$.

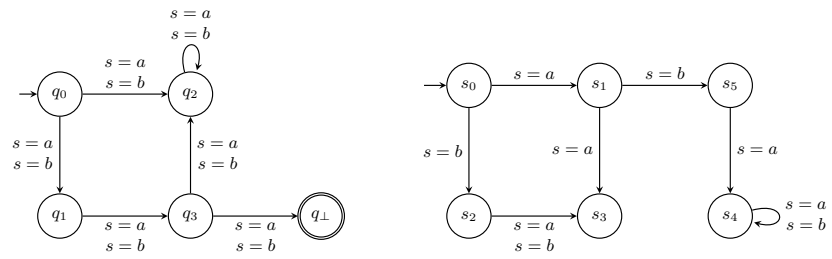
Definition 5. *Let Φ be a property. The* observation-point language *of Φ is:*

$$\mathcal{R}(\Phi) = \{u \in \mathsf{CIO}(U)^* \mid \exists z \cdot c' \in \mathsf{CIO}(O)^*.z \in \mathscr{L}(\Phi) \wedge z \cdot c' \notin \mathscr{L}(\Phi) \wedge |u| = |z \cdot c'|\}$$

To compute a CA with final states accepting $\mathcal{R}(\Phi)$ one takes the complement $\overline{\Phi}$ of Φ. The CA $\overline{\Phi}$ has one final state $q^R_\perp$ which is a sink. The CA with final states $\mathcal{R}$ accepting $\mathcal{R}(\Phi)$ is obtained from $\overline{\Phi}$ by removing the self loops in the sink state $q^R_\perp$ and by replacing every transition of $\overline{\Phi}$ with a transition between the same pair of states for every label $c \in \mathsf{CIO}(U)$. Then, to build a most general CA $\mathsf{MGU}(\mathcal{A}, \Phi)$ such that replacing $\mathcal{A}$ with $\mathsf{MGU}(\mathcal{A}, \Phi)$ is a correct update w.r.t. Φ for all contexts, one can proceed as follows.

1. **Determinize** $\mathcal{R}$ using the subset construction.
2. **Complete** $\mathcal{A}$ by adding a sink state $q^A_\perp$ and obtaining $\mathcal{A}_\perp$.
3. **Compute the product of $\mathcal{A}_\perp$ with** $\mathsf{Subset}(\mathcal{R})$ using the synchronous join operator to obtain $\mathcal{A}_\perp \bowtie_s \mathsf{Subset}(\mathcal{R})$.
4. **Remove observation states**, that is all states $(q^A_\perp, Q_R)$ such that $q^R_\perp \in Q_R$, and take the connected component including the initial state.
5. **Transform the result into a CA without final states** by dropping the distinction between final and non-final states.

$\mathsf{MGU}(\mathcal{A}, \Phi)$ can be computed in time which is a double exponential in the size of Φ and polynomial in the size of $\mathcal{A}$, where the two exponentials are due to the subset constructions.



(a) CA with final states for $\mathcal{R}(\varphi)$. **(b)** Most general scheduler.

Fig. 5. CAs of Example 5.

Theorem 9. *Let Φ be a property and $\mathcal{A}$ a CA. For the synchronous embedding, the most general CA such that replacing $\mathcal{A}$ with $\mathcal{B}$ is a correct update w.r.t. Φ and for all contexts is* $\mathsf{MGU}(\mathcal{A}, \Phi)$.

Example 5. Consider the property P1 represented by the CA Φ back in Figure 4a. By the above procedure we can first obtain the CA with final states for $\mathcal{R}(\Phi)$ in Figure 5a, and then the most general scheduler $\mathsf{MGU}(\mathcal{A}, \Phi)$ in Figure 5b, which makes the update correct in the synchronous case for every context and for the property Φ. We are left with two kinds of traces: traces with prefix $r = a, r = b, r = a$ that behave as $\mathcal{A}$ for the first 3 steps, and traces of length less than 3 that behave differently w.r.t. $\mathcal{A}$. This corresponds to the intuition that the property can only reject traces at step 3.

We now characterize the updates correct w.r.t. all the properties and all contexts. In this case there are strong requirements on the updates. To be correct for all contexts, the update needs to be correct for the context that reports every communication to the outside world. Since properties are sets of traces, the new component $\mathcal{B}$ should have at most the traces of $\mathcal{A}$. Indeed, this condition is necessary and sufficient, for both the synchronous and asynchronous embedding.

Theorem 10. *Let $\mathcal{A}$ be a CA. Any $\mathcal{B}$ such that $\mathscr{L}(\mathcal{B}) = \mathscr{L}(\mathcal{A})$ is a most general update such that replacing $\mathcal{A}$ with $\mathcal{B}$ is correct for all properties and all contexts.*

5 Conclusion and Related Work

We studied the problem of finding out whether an update replacing a component $\mathcal{A}$ with a component $\mathcal{B}$ in a given context $\mathcal{C}$ is correct w.r.t. a safety property Φ. We also characterized the updates correct in any context (for a given property), for any property (in a given context), and for any property in any context. In all the cases, we considered both synchronous and asynchronous composition.

While many approaches tackle system update [19], the problem of ensuring correctness of a system upon update has received scarce attention till now.

Approaches based on behavioral congruences, such as [8], allow one to prove the correctness of updates when a component is replaced by a syntactically

different, but semantically equivalent one. Our approach is more general, allowing one to replace a component with a semantically different one.

Some approaches, such as [13], focus exclusively on type safety, that rules out obviously wrong behaviors, but is insufficient for establishing that given properties are preserved. In [14], instead, a program transformation to combine a program and an update into a new program presenting all the behaviors corresponding to applying the update at any allowed point is presented. The key advantage of our approach is that we can deal with many updates at once by comparing them with the most general one. In [23], a modular model checking approach to verify adaptive programs is proposed. They decompose the model checking problem following the temporal evolution of the system, while we decompose the verification problem following the structure of the system.

A line of work [2, 11] uses choreographic descriptions to obtain correctness of the updates by construction. However, this kind of approach can only deal with a few fixed properties such as deadlock freedom, race freedom and orphan-message freedom. Another related approach is presented in [12], where behavioral types are used to ensure that running sessions are not interrupted, and that provided services are preserved. Our approach is much more flexible than the two last approaches since it considers any property expressible as a CA.

The work in [18] categorizes different kinds of reconfigurations in the context of Reo connectors. Our updates correct for any property (in a given context) are called contractive in [18], and a property for which an update is correct (in a given context) is called an invariant for the update. However, in [18], nothing is said about the requirements that an update must satisfy to be contractive or to have a given invariant: these problems have been solved by the present paper.

The work in [22] is related to ours from the technical point of view. In particular, it provides us the framework to solve Inequation (1). However, [22] does not provide a construction for building an actual automaton in our case, namely, for CAs with both finite and infinite traces. Also, [22] has a different aim, since it does not consider update at all. It highlights, however, a connection between update and another challenging problem: the automatic synthesis of systems from logical specifications. Polynomial algorithms for restricted classes of specifications have been identified [1, 7]. These results could be exploited both to make our approach more efficient and to extend it to properties that go beyond safety, like liveness and deadlock freedom. Another problem related to ours is supervisory control of discrete event systems (see, e.g., [4]). The main difference is in the composition mechanism, which features a feedback control loop and introduces latency, while this does not happen in our case.

The problem of characterizing correct updates can also be studied in other settings, and indeed we plan to consider some of them in future work. For instance one may consider more complex properties, as hinted at above, or more complex automata, like timed automata or general CAs where the set of data values can be infinite. Finally, we want to apply our technique to more abstract models, starting from the ones based on CAs, such as REO [3] and Rebeca [20].

References

1. Alur, R., La Torre, S.: Deterministic generators and games for LTL fragments. ACM Trans. Comput. Logic 5(1), 1–25 (2004)
2. Anderson, G., Rathke, J.: Dynamic software update for message passing programs. In: APLAS. LNCS, vol. 7705, pp. 207–222. Springer (2012)
3. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
4. Aziz, A., et al.: Supervisory control of finite state machines. In: CAV. LNCS, vol. 939, pp. 279–292. Springer (1995)
5. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by Constraint Automata. Sci. Comput. Program. 61(2), 75–113 (2006)
6. Baier, C., et al.: Formal verification for components and connectors. In: FMCO. LNCS, vol. 5751, pp. 82–101 (2008)
7. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. of Computer and System Sciences 78(3), 911–938 (2012)
8. Bonchi, F., Brogi, A., Corfini, S., Gadducci, F.: A behavioural congruence for web services. In: FSEN. LNCS, vol. 4767, pp. 240–256. Springer (2007)
9. Bresolin, D., Lanese, I.: Most general property-preserving updates (TR). `http://www.cs.unibo.it/~lanese/work/lata17-tr.pdf` (2016)
10. Chatterjee, K., Doyen, L.: Games with a weak adversary. In: ICALP, LNCS, vol. 8573, pp. 110–121. Springer (2014)
11. Dalla Preda, M., et al.: Dynamic choreographies: Safe runtime updates of distributed applications. In: COORDINATION. LNCS, vol. 9037, pp. 67–82 (2015)
12. Di Giusto, C., Pérez, J.A.: Disciplined structured communications with consistent runtime adaptation. In: SAC. pp. 1913–1918. ACM (2013)
13. Duggan, D.: Type-based hot swapping of running modules. Acta Inf. 41(4-5), 181–220 (2005)
14. Hayden, C.M., et al.: Specifying and verifying the correctness of dynamic software updates. In: VSTTE. LNCS, vol. 7152, pp. 278–293. Springer (2012)
15. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing - degrees, models, and applications. ACM Comput. Surv. 40(3) (2008)
16. Lange, M.: Weak automata for the linear time μ-calculus. In: VMCAI. LNCS, vol. 3385, pp. 267–281. Springer (2005)
17. Leite, L.A.F., et al.: A systematic literature review of service choreography adaptation. Service Oriented Computing and Applications 7(3), 199–216 (2013)
18. Oliveira, N., Barbosa, L.S.: On the reconfiguration of software connectors. In: SAC. pp. 1885–1892. ACM (2013)
19. Seifzadeh, H., Abolhassani, H., Moshkenani, M.S.: A survey of dynamic software updating. J. of Software: Evolution and Process 25(5), 535–568 (2013)
20. Sirjani, M., et al.: Compositional semantics of an actor-based language using constraint automata. In: COORDINATION. LNCS, vol. 4038, pp. 281–297 (2006)
21. Tsai, M., Tsay, Y., Hwang, Y.: GOAL for games, omega-automata, and logics. In: CAV. LNCS, vol. 8044, pp. 883–889. Springer (2013)
22. Villa, T., et al.: The Unknown Component Problem - Theory and Application. Springer (2012)
23. Zhang, J., Goldsby, H., Cheng, B.H.C.: Modular verification of dynamically adaptive systems. In: AOSD. pp. 161–172. ACM (2009)