Adaptor synthesis for real-time components[⋆]

Massimo Tivoli[1], Pascal Fradet[2], Alain Girault[2], and Gregor Goessler[2]

[1] University of L'Aquila[⋆⋆]
Dip. Informatica, via Vetoio 1, 67100 L'Aquila, Italy
tivoli@di.univaq.it
[2] INRIA Rhône-Alpes - POP ART project
655 avenue de l'Europe, 38330 Montbonnot, France
{Pascal.Fradet, Alain.Girault, Gregor.Goessler}@inrialpes.fr

Abstract. Building a real-time system from reusable or COTS components introduces several problems, mainly related to compatibility, communication, and QoS issues. We propose an approach to automatically derive adaptors in order to solve black-box integration anomalies, when possible. We consider black-box components equipped with an expressive interface that specifies the interaction behavior with the expected environment, the component clock, as well as latency, duration, and controllability of the component's actions. The principle of adaptor synthesis is to coordinate the interaction behavior of the components in order to avoid possible mismatches, such as deadlocks. Each adaptor models the correct assembly code for a set of components. Our approach is based on labeled transition systems and Petri nets, and is implemented in a tool called SynthesisRT. We illustrate it through a case study concerning a remote medical care system.

1 Introduction

Due to their increasing complexity, control systems are nowadays often designed in a modular approach by means of libraries of building blocks. This has lead to a need of a component-based approach for building real-time systems out of a set of already implemented components. Building a real-time system from reusable or *Commercial-Off-The-Shelf* (COTS) components introduces several problems, mainly related to compatibility, communication, and quality of service (QoS) issues [2, 10–12, 18]. Indeed, incompatibilities between the components may arise and make their composition impossible.

In this paper, we show how to deal with these problems within a lightweight component model where components follow a data-flow interaction model. Each component declares input and output ports which are the points of interaction with other components and/or the execution environment. Input (resp., output) ports of a component are connected to output (resp., input) ports of a different component through synchronous links. In our framework, a component interface includes a formal description of the *interaction protocol* of the component with

[⋆] This work has been partially funded by the Alidecs project and the Artist II European network of excellence.

[⋆⋆] This work has been done while the first author was a postdoctoral fellow in the Pop Art project team at INRIA Rhône-Alpes.

its expected environment in terms of sequences of writing and reading actions to and from ports. The interface language is expressive enough to specify QoS constraints such as writing and reading *latency*, *duration*, and *controllability*, as well as the component's *clock* (*i.e.,* its activation frequency). In order to deal with incompatible components (*e.g.,* clock inconsistency, read/write latency/duration inconsistency, mismatching interaction protocols, *etc.*) we synthesize component *adaptors* interposed between two or more interacting components. An adaptor is a component that mediates the interaction between the components it supervises, in order to harmonize their communication. Each adaptor is automatically derived by taking into account the interface specification of the components it supervises. The adaptor synthesis allows the developer to automatically and *incrementally* build *correct-by-construction* systems from third-party components, hence reducing time-to-market and improving reusability. The space complexity of the synthesis algorithm is exponential in the number of states of the automaton modeling the interaction protocol of each component. Thus, incrementality is crucial to manage the complexity of real systems.

We have formalized the adaptor synthesis algorithm by using Petri nets [16] theory, and we address its correctness in a companion paper [19]. Moreover, in order to realize the whole approach, we have implemented a tool, called *Synthesis Real Time* (SynthesisRT) [19], which we have used on a case study concerning a remote medical care system (RMCS).

The remainder of the paper is organized as follows. Section 2 introduces the notions of latency, duration, controllability, and local/global time/clock. Section 3 provides an informal overview of our method. Section 4 presents our component specification formalism and its semantics in terms of *Labeled Transition Systems* (LTSs). Section 5 formalizes the technical core of adaptor synthesis. Section 6 describes our method at work on the RMCS case study. Finally, Section 7 summarizes our work and presents related work and future extensions.

2 Background

In this section, we introduce the background notions used by our framework.

2.1 Context

We want to build component-based real-time systems by assembling third-party black-box components. Black-box means that the component source code is not available to the system designer. Each component is equipped with a rich interface that describes its behavior as well as real-time properties. According to the "*design by contract*" approach [18], such an interface specification is given by the component developer, who is aware of the information needed. An interface includes:

– A behavioral interface specification. This specification is given in terms of a *Labeled Transition System* (LTS) that models the sequences of actions that the component performs when it interacts with its environment. As it is explained below, this LTS contains also timing information.

- A periodic clock that, for reuse purposes, is instantiated at assembly-time. It specifies a sequence of instants by an infinite stream of boolean values: 1 denotes an instant where the component is *enabled* (it can perform an action or let the time elapse) and 0 denotes an instant where then component is *disabled*. A periodic clock can be finitely represented by its periodic sub-stream (*e.g.,* the clock $(10)^{\omega}$ represents the infinite stream 10101010101010...). The global time is defined by the clock $(1)^{\omega}$ that is called the *base clock*. The clock of each component defines a time that is *local* to the component. It characterizes the component speed and can be seen as a sub-clock of the base clock. For hierarchies of components, the local clock of each component is a sub-clock of the clock of its super-component. We refer the reader to [5] for a comprehensive presentation of the periodic clock concept.
- A latency (a natural number) for each action. It specifies the number of *global* time units that can pass before the action is performed. In other words, the component may choose to synchronize with its environment to perform the corresponding action any time before the latency is elapsed.
- A duration (an interval of natural numbers) for each action. It specifies the *local* time units needed for the action execution. For instance, a duration $[1, 2]$ indicates that the action may require one or two instants where the component is enabled to complete. Contrary to the latency, the precise duration cannot be chosen. The component must synchronize correctly with its environment for every possible execution time specified by its duration.
- A controllability tag for each action. An uncontrollable action (*i.e.,* tagged with u) cannot be disabled. For example, inputs coming from a sensor are often considered as uncontrollable since they must be accepted and treated by the component. In contrast, controllable actions (without a tag) can be safely disabled (*e.g.,* by a supervisor or an adaptor), for instance to prevent a deadlock.

2.2 Architectural model

In this section, we provide an overview of our architectural model using a small example. Figure 1 shows the architectural specification of a black-box component C_1, with a clock port w_1, which interacts with its environment through the input port a and the output port b.

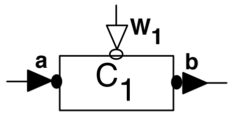

Fig. 1. Architectural schema of component C_1

In general, a component can have several input and output ports. Components are connected to each other through their ports and interact *synchronously*. An input port of a component can be connected to an output port of a different component. Input (resp., output) ports support a *reading* (resp., *writing*) operation. Synchronous communication implies that reading and writing operations among connected ports are blocking actions. In other words, connected components are forced to synchronize on complementary read/write operations. *E.g.*, let the input port p_1 and the output port p_2 be connected: a reading from p_1 has to synchronize with a writing to p_2. This style of communication is not a limitation because it is well known that, with the introduction of a buffer component, we can always simulate an asynchronous system by a synchronous one [13].

The clock port of a component can be seen as a special input port whose current value (either 1 or 0) depends on the periodic clock that has been assigned to the component and on the current instant of the global time. It is not connected to other ports since it serves only to assign a periodic clock to the component at assembly-time.

3 Overview

In this section, we informally describe the main steps of our method as illustrated in Figure 2. Although we took inspiration from [3], our synthesis algorithm is very different from theirs as it is discussed in Section 7.

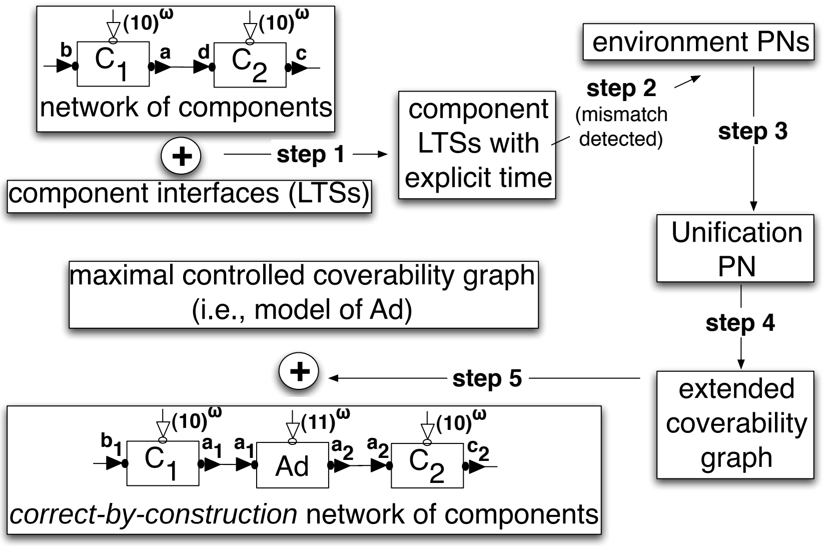
(10)^ω
(10)^ω
b
C1
a
d
C2
c
network of components
+
step 1
component interfaces (LTSs)
component LTSs with explicit time
step 2
(mismatch detected)
environment PNs
step 3
Unification PN
step 4
extended coverability graph
step 5
maximal controlled coverability graph (i.e., model of Ad)
+
(10)^ω
(11)^ω
(10)^ω
b1
C1
a1
a1
Ad
a2
a2
C2
c2
correct-by-construction network of components

Fig. 2. Main steps of adaptor synthesis for real-time components

We take as input the architectural specification of the network of components to be composed and the component interface specifications. The behavioral models of the components are generated in form of LTSs that make the elapsing of time explicit (step 1). Connected ports with different names are renamed such that complementary actions have the same label in the component LTSs (see actions a and d in Figure 2). Possible mismatches/deadlocks are checked by looking for possible sink states into the parallel composition of the LTSs. The adaptor synthesis process starts only if such deadlocks are detected.

The synthesis first proceeds by constructing a Petri net (PN) representation of the environment expected from a component in order to avoid deadlocks (step 2). It consists in complementing the actions in the component LTSs that are performed on connected ports, considering the actions performed on unconnected ports as internal actions. A buffer storing read and written values is modeled as a place in the environment PN for each IO action. Each such PN represents a partial view of the adaptor to be built. It is partial since it reflects the expectation of a single component. In particular, a write (resp. read) action gives rise to a place (buffer) without outgoing (resp. incoming) arcs.

The partial views of the adaptor are composed together by building causal dependencies between the reading/writing actions and by unifying time-elapsing transitions (step 3). Furthermore, the places representing the same buffer are merged in one single place. This *Unification PN* desynchronizes emission from reception using buffers. However, the unification PN may include behaviors that deadlock and/or require unbounded buffers. In order to obtain the most permissive and correct adaptor, we generate an extended version of the graph usually known in PNs theory as the coverability graph [8] (step 4).

Our method automatically restricts the behavior of the adaptor modeled by the extended coverability graph in order to keep only the behaviors that are deadlock-free and that use finite buffers (*i.e.,* bounded interactions). This is done by automatically constructing, if possible, an "instrumented" version of our extended coverability graph, called the *Controlled Coverability Graph (CCG).* The CCG is obtained by pruning from the extended coverability graph both the *sinking* paths and the *unbounded* paths, by *controller synthesis* [17] (step 5). This process also performs a *backwards propagation* in order to correctly take into account the case of sinking and unbounded paths originating from the firing of uncontrollable transitions.

If it exists, the maximal CCG generated is the LTS modeling the behavior of the correct (*i.e.,* deadlock-free and bounded) adaptor. This adaptor models the correct-by-construction assembly code for the components in the specified network. If it does not exist, a correct adaptor assembling the components given as input to our method cannot be derived, and hence our method does not provide any assembly code for those components.

4 The interface specification and its translation

In this section, we present the interface specification language by continuing the small example introduced before (the component C_1 described in Section 2.2). We have defined a higher-level language, called DLiPA [19], based on process algebra. In this paper, we start from an LTS, a form that DLiPA processes can be easily translated into.

Our source LTSs are labeled with actions of the form $x_l^{\{u\}\ [i,j]}$ where x denotes the action (read or write), l its allowed latency, $[i,j]$ its duration, and u, if present, the uncontrollability of the action.

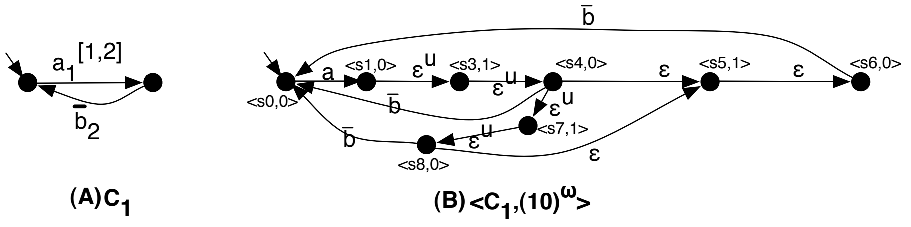

Fig. 3. (A) Behavioral interface of C_1 and (B) its semantic model with respect to $(10)^\omega$

Figure 3.(A) gives the interface specification of the component C_1 as an LTS. From its initial state (denoted by an incoming arrow without source state), C_1 performs an action a (*i.e.,* it reads from port a) followed by an action $\bar{b}$ (*i.e.,* it writes to port b) that returns to the initial state. All C_1 actions are controllable (no u tag). The action a has latency 1, *i.e.,* its execution can be delayed by one global time unit at most. Moreover, a has duration $[1,2]$ meaning that its execution can take either one or two local time units. Similarly, the execution of $\bar{b}$ can be delayed by two global time units at most and takes no time.

Figure 3.(B) presents the semantic model of C_1 that has been derived by taking into account the interface specification of Figure 3.(A) and a periodic clock, here $(10)^\omega$, which has been assigned to C_1. This semantics is noted $\langle C_1, (10)^\omega\rangle$. It is an LTS modeling the interaction behavior of C_1 with its expected environment and making time elapsing explicit. The clock $(10)^\omega$ has been assigned by the designer of the system to be assembled and it represents the required component activation frequency. The LTS of $\langle C_1, (10)^\omega\rangle$ is produced by compiling latency and duration information into *abstract* actions ε representing time elapsing. Each state of the LTS is named by a pair made of a label (*e.g.,* $s0$) and a global time instant (*e.g.,* 0). These instants refer to the finite representation of the assigned periodic clock, *i.e.,* they are the instants $0,\ldots,l-1$ where l is the length of the clock's period. In our example, where the clock is $(10)^\omega$, the instant 0 represents instants where C_1 is enabled (*i.e.,* it can perform some action or let the time elapse) whereas the instant 1 represents instants where C_1 is disabled (*i.e.,* it can only let the time elapse).

A transition labeled by a concrete action (*e.g.,* a) is instantaneous: it represents the starting point for the execution of the action. For example, the transition $\langle s0,0\rangle \xrightarrow{a} \langle s1,0\rangle$ in Figure 3.(B) means that C_1 starts to read from port a. A transition labeled by an abstract action ε or ε^u lets the time elapse: it represents a tick of the global clock (*e.g.,* $\langle s1,0\rangle \xrightarrow{\varepsilon^u} \langle s3,1\rangle$ in Figure 3.(B)).

Latency is translated using the controllable action ε. For instance an action x with latency 1 is translated into two sequences of transitions: one sequence performing x immediately and another sequence performing x after an ε-transition. If one branch leads to a deadlock, the environment (*i.e.,* the adaptor to be synthesized) may choose the other one by synchronizing only with it.

Duration is translated using the uncontrollable action ε^u. For instance, assuming the clock $(1)^\omega$, an action x with duration $[1, 2]$ is translated into the transition x followed by the branching between one or two ε^u-transitions. The uncontrollability enforces the composition with the environment to be compatible with both time-elapsing possibilities. Note that, since duration refers to local time and the semantics refers to the global time, the previous example with a clock $(10)^\omega$ would be translated into the transition x followed by the branching between two or four ε^u-transitions depending on the clock instant (assuming the action x is enabled initially).

In the LTS of Figure 3.(B) (*i.e.,* $\langle C_1, (10)^\omega\rangle$), a duration unit is represented by two ε^u-transitions. Note also that the local clock influences the actual latency. For instance, according to clock $(10)^\omega$, C_1 either executes b immediately (from the time it is enabled) or waits exactly two global time units to execute it: a one time unit wait leads to a state where the component is disabled and b cannot be performed. Analogously, in order to represent the latency of a, an ε-transition should be produced from the initial state. However, this transition is pruned since it is controllable and leads to a sink state (only a read from a is permitted but it is disabled).

To define the semantics of a system (*i.e.,* a network of components), we put in parallel the semantic models of the components by forcing the synchronization on complementary concrete actions and on abstract actions. Components synchronize *pairwise* on complementary concrete actions by producing, for each synchronizing pair $b/\bar{b}$, a τ-transition at the level of the composed system, where τ denotes an internal action. Components synchronize, *altogether*, on time-elapsing transitions by producing a time-elapsing transition at the level of the system. Whenever two or more components have a mismatching interaction due to some behavioral inconsistency, a deadlock occurs in the composed system (*i.e.,* a sink state is produced in the LTS of the system). This is precisely what we avoid thanks to our adaptor synthesis method, presented in the next section. We refer to [19] for further details.

5 Adaptor synthesis

In this section, we illustrate our method using another small example and formalize part of it. For space reasons, we focus only on the formalization of the Unification PN (see Definition 1) and we omit other formal details that will be illustrated through the explanatory example.

5.1 Unification PN generation

Let us suppose that the designer wants to build an assembly S formed by two components C_1 and C_2 whose semantic models are shown in Figure 4.

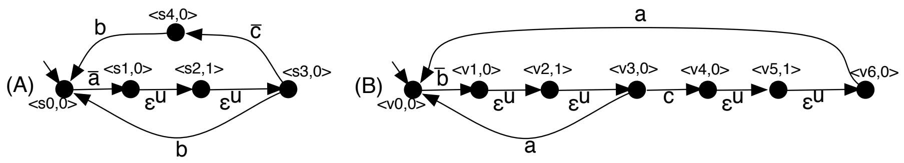

Fig. 4. ***After step 1:*** (A) $\langle C_1, (11)^\omega \rangle$; (B) $\langle C_2, (10)^\omega \rangle$

Note that the periodic clocks of C_1 and C_2 have the same length. This is required in order to perform the generation of the Unification PN. This requirement is not a limitation since, although the designer can specify clocks with different length, they can be always rewritten in such a way that they have the same length by taking the least common multiple of the different lengths. In Figures 5.(A) and 5.(B) we show respectively the PNs modeling the environment expected from $\langle C_1, (11)^\omega \rangle$ and $\langle C_2, (10)^\omega \rangle$ in order not to block.

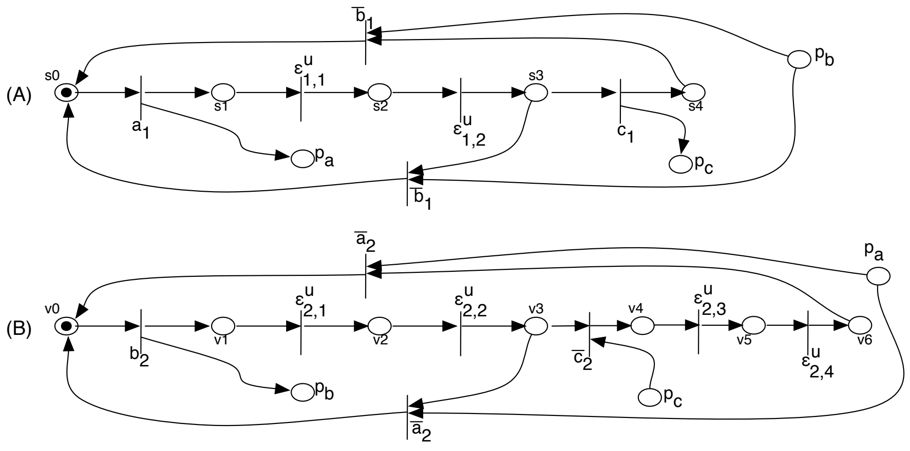

Fig. 5. ***After step 2:*** Component PNs - (A) PN_1; (B) PN_2

For technical reasons, the actions have been relabeled. Since, now, all the latencies and durations have been made explicit through ε-transitions, the indexing that has been used for the action labels must not be confused with the one used above to specify the latency. We recall that each environment PN is a partial view of the adaptor to be built since it reflects the expectation of only one component. In particular, for each state in the component LTS, there is a place in the environment PN. The initial marking puts a token in the place corresponding to the initial state. For each transition labeled with a concrete action in the component LTS, there is a transition labeled with the *complementary* action in the environment PN. The transition label is such that it contains the information concerning which component has performed the corresponding action (through a suitable indexing: *e.g.*, subscript 1 for C_1 and 2 for C_2).

For each component writing action to an output port x, a place p_x is produced and an arc from the corresponding transition to p_x is added. It corresponds to the fact that, in order to synchronize with a component, the adaptor reads and stores values into an internal buffer. A stored value will be written as output as soon as the adaptor synchronizes with a component that expects to read this value. Component reading actions are handled in a complementary way. In this way, the adaptor desynchronizes the received events from their emission, hence solving mismatches arising from the fact that different components perform complementary actions at different instants.

For a time-elapsing transition in the component LTS, the corresponding transitions, places, and arcs are generated in the environment PN as it is shown in Figure 5. That is, a correct environment for a component has to let the time elapse whenever the component lets the time elapse as well.

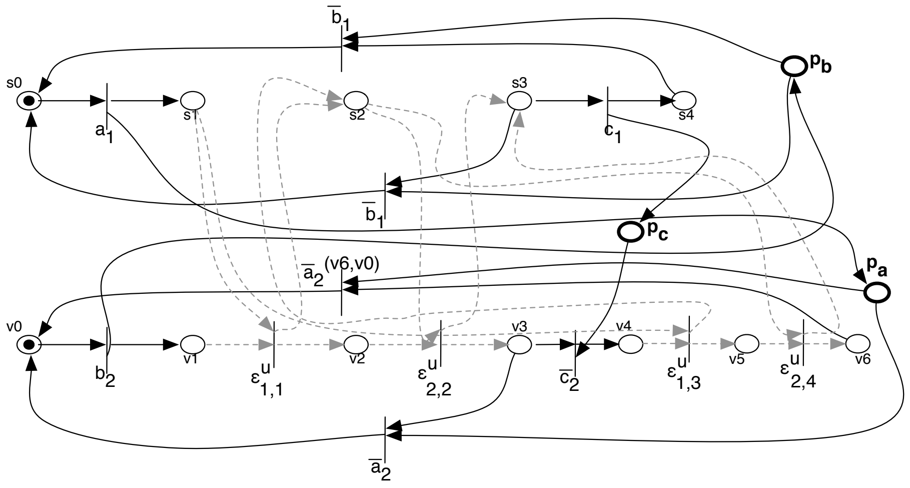

Fig. 6. ***After step 3:*** $PN_{1,2}$: the Unification PN for PN_1 and PN_2

Actions that do not force the component to synchronize with the environment can be freely performed; the adaptor must not preempt them and produces an internal action whenever they occur (there is no such action in our example). We refer the reader to [19] for a formal definition of environment PN.

After the partial views of the adaptor have been built, they are composed in order to obtain the Unification PN. In Figure 6 we show the Unification PN (*i.e.*, $PN_{1,2}$) that has been obtained after the unification of PN_1 and PN_2. The Unification PN $PN_{1,2}$ is automatically derived from the union of PN_1 and PN_2 plus a unification operation of their time-elapsing transitions. Informally, casual dependencies between the reading and writing of data are generated by performing the union of the sets of places, arcs, and transitions, except for the arcs and transitions concerning the elapsing of time. Time-elapsing transitions are composed using the synchronous product. Figure 6 shows the obtained time-elapsing transitions as dashed and grey arrows. For readability issues, we have drawn only the fireable transitions. For example, the first time-elapsing transition of PN_1 composed with the first time-elapsing transition of PN_2 is fireable. Note that the first time-elapsing transition of PN_1 composed with the third time-elapsing transition of PN_2 is fireable as well (after PN_1 has performed one loop). The step to derive the unification PN is formalized by Definition 1:

Definition 1 (Unification PN). *Let $PN_i = (P_i, T_i, F_i, M_0^i)$ (where $i = 1, \ldots, n$, P_i is the set of places, T_i is the set of transitions, F_i is the set of arcs, and M_0^i is the initial marking) be the PN modeling the environment expected from the component C_i. The* Unification PN *is the Petri Net $UPN = (P, T, F, M_0)$, where:*

- *$P = \bigcup_{i=1}^{n} P_i$;*
- *$T = \bigcup_{i=1}^{n} T_i' \cup \{\varepsilon^u_{k_1,...,k_n} \mid \forall i = 1, ..., n \ . \ \varepsilon^x_{i,k_i} \in T_i \wedge \exists i \ . \ \varepsilon^u_{i,k_i} \in T_i\} \cup \{\varepsilon_{k_1,...,k_n} \mid \forall i = 1, ..., n \ . \ \varepsilon_{i,k_i} \in T_i\}$, where T_i' is T_i without time-elapsing transitions and the superscript x is either equal to 'u' or is empty;*
- *$F = \bigcup_{i=1}^{n} F_i' \cup \bigcup_i \{(p, \varepsilon^u_{k_1,...,k_n}), (\varepsilon^u_{k_1,...,k_n}, p') \mid p, p' \in P_i \wedge (p, \varepsilon^x_{i,k_i}) \in F_i \wedge (\varepsilon^x_{i,k_i}, p') \in F_i \wedge \varepsilon^u_{k_1,...,k_n} \in T\} \cup \{(p, \varepsilon_{k_1,...,k_n}), (\varepsilon_{k_1,...,k_n}, p') \mid p, p' \in P_i \wedge (p, \varepsilon_{i,k_i}) \in F_i \wedge (\varepsilon_{i,k_i}, p') \in F_i \wedge \varepsilon_{k_1,...,k_n} \in T\}$, where F_i' is F_i without arcs to or from time-elapsing transitions, and the superscript x is either equal to 'u' or is empty;*
- *for each $p \in P$ if $\exists i. M_0^i(p) = 1$ then $M_0(p) = 1$, otherwise $M_0(p) = 0$.*

The following is an upper-bound estimation of the size of the Unification PN in terms of its number of places and transitions:

$|P| = \sum_{i=1}^{n} |P_i|$
$|T| = \sum_{i=1}^{n} |T_i'| + \prod_{i=1}^{n} |T_i^{te}|$ where T_i^{te} is the set of time-elapsing transitions of PN_i

Note that the number of places and immediate (*i.e.*, non time-elapsing) transitions of the Unification PN grows up linearly with respect to the number of places and immediate transitions of the component PNs; whereas the number of time-elapsing transitions is exponential with respect to the number of time-elapsing transitions of the component PNs.

5.2 Controlled Coverability Graph synthesis (CCG)

After the Unification PN has been generated, its maximal CCG is automatically derived, if it exists. We first generate the extended coverability graph of the Unification PN. Given a PN (P, T, F, M_0), we construct the marking graph in the standard way. From M_0, we obtain as many markings as the number of the enabled transitions. From each new marking, we can again reach more markings. This process results in a graph representation of the markings. Nodes represent markings generated from M_0 (the initial node) and its successors, and each arc represents a transition firing, which transforms one marking into another. However, the above representation will grow infinitely large if the PN is unbounded. To keep it finite, we introduce a special symbol ω to indicate a possibly infinite number of tokens in some place. ω can be thought of as "infinity". It has the properties that for each integer n, $\omega > n$, $\omega \pm n = \omega$, and $\omega \geq \omega$. Given markings M and M' such that: (1) M' is reachable from M, and (2) $\forall p, M'(p) \geq M(p)$ (*i.e.*, M is coverable by M'), then, for each place q such that $M'(q) > M(q) \geq 1$, we replace $M'(q)$ by ω in the extended coverability graph. This is the same criterion as the *termination criterion* used by Cortadella et al. to identify *irrelevant markings* [6]. They conjecture that this criterion is *complete* [6], meaning that if a bounded and non-blocking execution exists, it will be represented in the extended coverability graph.

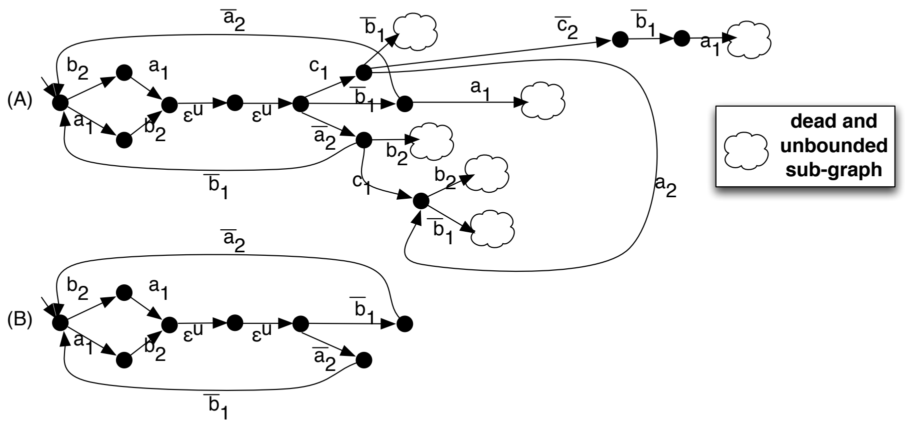

Fig. 7. ***After step 4:*** (A) extended coverability graph of $PN_{1,2}$; ***After step 5:*** (B) its *controlled* version

By continuing our example, we partially show the extended coverability graph of $PN_{1,2}$ in Figure 7.(A). The cloud-nodes are portions of the coverability graph made only of paths whose nodes are either *dead* or contain *unbounded* markings. Informally, a dead (resp., unbounded) marking is a node without successors (resp., that represents a marking in which some place has stored a potentially infinite number of tokens) or whose successors always lead to dead (resp., unbounded) markings. We refer to [19] for a formal definition of dead and unbounded markings, and of CCG.

In Figure 7.(B), we show the maximal CCG of $PN_{1,2}$. The maximal CCG is the most permissive one among all possible CCGs. Informally, it is obtained from Figure 7.(A) by pruning the transitions that lead inevitably to cloud-nodes and that are controllable. The pruning of controllable transitions, as well as the "most permissive" notion, is borrowed from Discrete Controller Synthesis [17].

6 Case study: a remote medical care system

We now apply our approach to a case study, borrowed with minor modifications from [4]: a *Remote Medical Care System* (RMCS). The RMCS provides monitoring and assistance to disabled people. A typical service is to send relevant information to a local phone-center so that medical or technical assistance can be timely notified of critical circumstances. The RMCS can be built from eight

COTS components (*Alarm*, *Line*, *Control*, *RAlarm*, etc.) assembled into three composite components: *User*, *Router*, and *Server* (see Figure 8). Using our adaptor synthesis method and its associated tool (SynthesisRT), it has been possible to incrementally and automatically assemble a correct by construction RMCS.

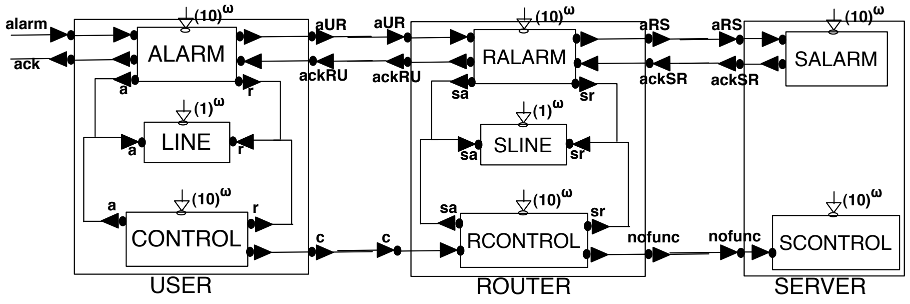

Fig. 8. The software architecture of the RMCS

When a patient needs help (*i.e.*, the uncontrollable signal *alarm* is emitted), *User* sends either an alarm ($\overline{aUR}$) or a check message ($\overline{c}$) to *Router*. After sending an alarm, *User* waits for an acknowledgment (*ackRU*) and indicates the conclusion of the alarm dispatching to the patient (*ack*). *Router* waits for check or alarm messages from *User* (*c* or *aUR*). It forwards alarm messages to *Server* (*aRS*) and checks the state of *User* through the check message (*c*). *Server* dispatches the alarm requests (*aRS*).

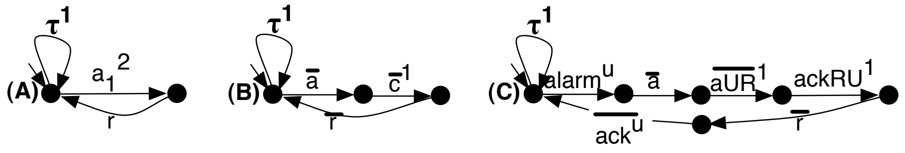

Fig. 9. Behavioral specification of (A) *Line*, (B) *Control*, and (C) *Alarm*

Router and *Server* are connected through a dedicated line (modeled by the component *SLine*) that is always available. Conversely, *User* and *Router* are connected through a usual phone line (modeled by the component *Line*) that can be busy.

For space reasons, we only show the part of the case study that concerns the assembly of the correct-by-construction version of *User*. We refer to [19] for a complete description of both the case study and our SynthesisRT tool. *User* models the logic of the control device provided to patients in order to dispatch alarms. It is an assembly of the three components *Control*, *Alarm*, and *Line*. Figure 9 provides the interface specifications of these components. From these behavioral specifications, SynthesisRT automatically derives the corresponding LTSs. Then, the CADP toolbox [9] is used to derive the LTS representing *User*[3]. CADP allows us to detect possible deadlocks and to exhibit deadlocking traces. For instance, in *User*, an alarm request can deadlock whenever *Alarm* receives

[3] Referring to Figure 2, steps 1, 3, and 5 are performed with SynthesisRT, step 2 with CADP, and step 4 with TINA.

an alarm request from the patient and *Control* gets the *Line* to send a check message to *Router*. Figure 10 represents a deadlocking trace where, after an alarm request, *Control* and *Line* synchronize (producing a τ) and let time elapse (*i.e.*, perform a and $\overline{c}$) whereas *Alarm* is still waiting on action $\overline{a}$ that should be performed immediately (no latency).

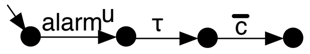

Fig. 10. A deadlocking trace of *User*

An adaptor is therefore required to avoid deadlocks in *User*. SynthesisRT automatically derives the environment PNs of *Line*, *Control*, and *Alarm*, as well as their Unification PN. The Unification PN is encoded in a file that can be fed to the TINA tool [1]. TINA is used to automatically derive the extended coverability graph of the generated Unification PN. The coverability graph is generated in 0.061 seconds on a Macbook Pro; it is unbounded and has 348 states, 763 transitions, 197 unbounded markings/states and no dead marking/state. This means that the message reordering has been sufficient to solve the detected deadlock, but it can still lead to some buffer overflows.

At this point, SynthesisRT is used again to automatically derive, from the uncontrolled coverability graph, its corresponding maximal CCG that prevents the reaching of the unbounded states. The maximal CCG is the LTS of the synthesized adaptor (Ad_{user}) that allows one to correctly assemble *User*. In our example, the adaptor is generated in 0.127 seconds but it is too large to be presented here; it has 116 states and 242 transitions. The deadlock is solved by Ad_{user} using message buffering and reordering. More precisely, when *Line* and *Control* perform a and $\overline{c}$, Ad_{user} synchronizes with *Alarm* on the line request $\overline{a}$. It stores the received request in a buffer in order to forward it when the line is released by *Control*. Then, the execution of *Alarm* can proceed and reach a point where it can let the time elapse, as required by *Control* and *Line*.

So, by putting Ad_{user} in parallel with *Line*, *Control*, and *Alarm*, we obtain the correct-by-construction version of the composite component *User*. We have also used SynthesisRT to derive three other adaptors: Ad_{router} (interposed between *RAlarm*, *SLine*, and *RControl*), Ad_{rs} (interposed between the adapted router and *Server*) and Ad_{rmcs} (interposed between the adapted composite router/server component and the adapted user component). Through the synthesis of these four adaptors, we have incrementally and automatically built a correct-by-construction RMCS.

7 Conclusion

In this paper, we have described an adaptor-based approach to assemble correct by construction real-time components that take into account interaction protocols, timing information, and QoS constraints. Our approach focuses on detection, correction, and prevention of deadlocks and unbounded buffers due to mismatching protocols. The main idea is to build a model of the environment of the component and to extract a controlled version (an adaptor) preventing deadlocks and unbounded buffers. In the general case, the space complexity of the synthesis algorithm is exponential in the number of states of the component LTSs. We have validated the approach by means of a case study.

Our work is related to several techniques in different research areas. In control theory, a related technique is *discrete controller synthesis* [17]. The objective is to restrict the system behavior so that it satisfies a *specification.* This is achieved by automatically synthesizing a suitable controller w.r.t. the specification. Beyond restricting the system behavior, our approach also extends it to resolve possible mismatches. For instance, while the approach in [17] performs only deadlock prevention, our approach performs also deadlock correction.

Another related work in synchronous programming is the synchronizing of different clocks. In [5], each input and output port is associated with a periodic clock. Adaptation is performed at the level of each connection between ports using finite buffers. It is sufficient to look at the clocks of two connected ports and to introduce a delay by interposing a *node buffer* between the two ports. In our context, adaptation must be performed at the component level by taking into account several dimensions of the specification: the component clock, the interaction protocol, the latency, duration, and controllability of each action. For this reason, introducing delays is not sufficient and, *e.g.*, the reordering or inhibition of actions may be necessary.

Related work in *interface automata* theory [7] also uses LTSs to model the input/output behavior of components. When composing two LTSs, they derive a constraint on their environment such that deadlocks are avoided, but they do not produce an adaptor to solve the incompatibilities between the two components.

Related work in component adaptation [3] and component interface compatibility [15] has shown how to automatically generate the behavioral model of an adaptor from: (i) a partial specification of the interaction behavior of the components and (ii) an abstract specification of the adaptor. In contrast with our work, they do not deal with real-time attributes. Although we took inspiration from [3] with respect to the PN encoding into the TINA tool and the use of CADP, our synthesis algorithm is very different from theirs since they do not have to take into account time-elapsing actions. Moreover, both techniques in [3] and [15] consider all component actions to be controllable, and neither considers the problem of synthesizing an adaptor model that ensures to always have bounded buffers.

Our approach focuses only on the automatic generation of the behavioral model of the correct adaptor. Future work shall consider the generation of the adaptor's actual code using, *e.g.*, synchronous languages such as Signal, Lustre, or Esterel. So far, the clocks are fixed before synthesizing the adaptor. Changing a component clock means re-executing the synthesis algorithm. An interesting extension would be to automatically derive clock-independent adaptors. A component clock would become a function of the adaptor clock. When the adaptor clock is instantiated, the component clocks will be instantiated as well to obtain a correct-by-construction assembly. Another possible future work is to study and formalize component architectures for which incremental adaptor synthesis is equivalent to a centralized adaptor synthesis.

References

1. B. Berthomieu, P. Ribet, and F. Vernadat. Construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42(14), 2004. TINA web page: `http://www.laas.fr/tina/`.
2. B. Boehm and C. Abts. COTS integration: Plug and pray? *IEEE Computer*, 32(1), 1999.
3. C. Canal, P. Poizat, and G. Salaün. Synchronizing behavioural mismatch in software composition. In *FMOODS*, volume 4037 of *LNCS*, 2006.
4. M. Cioffi and F. Corradini. Specification and analysis of timed and functional TRMCS behaviors. In *Proc. of the 10th IWSSD*, 2000.
5. A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. Synchronization of periodic clocks. In *Proc. of the 5th EMSOFT*, 2005.
6. J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Wanatabe. Quasi-static scheduling of independant tasks for reactive systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1492–1514, Oct. 2005.
7. L. de Alfaro and T. Henzinger. Interface automata. In *Annual Symposium on Foundations of Software Engineering, FSE'01*, pages 109–120. ACM, 2001.
8. A. Finkel. The minimal coverability graph for Petri nets. In *Proc. of the 12th APN*, volume 674 of *LNCS*, 1993.
9. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *EASST Newsletter*, 4, 2002. `http://www.inrialpes.fr/vasy/cadp`.
10. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6), 1995.
11. P. Inverardi, D. Yankelevich, and A. Wolf. Static checking of system behaviors using derived component assumptions. *ACM TOSEM*, 9(3), 2000.
12. N. Kaveh and W. Emmerich. Object system. *8th FSE/ESEC*, 2001.
13. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
14. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 1989.
15. R. Passerone, L. de Alfaro, T. Henzinger, and A. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *ICCAD*, 2002.
16. C. Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, 1962.
17. P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 1(77), 1989.
18. C. Szyperski. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, 1998.
19. M. Tivoli, P. Fradet, A. Girault, and G. Gössler. Adaptor synthesis for real-time components. Research report, INRIA, 2007, to appear.