Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 295 (2013) 11–29

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

From Timed Reo Networks to Networks of Timed Automata

Natallia Kokash[1], Mohammad Mahdi Jaghoori[2], Farhad Arbab[3]

Science Park 123, 1098XG
Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

Abstract

The Reo coordination language is an extensible graphical notation for component or service coordination wherein independent autonomous software entities exchange data through a connector or a network imposing synchronization and data constraints on those entities. Each connector is formed from a set of binary connectors, called channels, with precise semantics and, thus, amenable to formal verification. However, the development of verification tools for Reo-specific semantic models, namely, constraint automata with its multiple extensions to represent quality of service, time constraints, context-dependent or probabilistic behavior would require years of research and development. A much more promising approach is to exploit already existing verification tools. In this paper, we present a mapping of timed Reo networks to networks of timed automata used for system specification in Uppaal. Uppaal is a state-of-the-art toolset for modeling, validation and verification of real-time systems used in many large-scale industrial projects. Our work enables its application to the compositional analysis of timed service-based workflow models specified with Reo.

Keywords: Service Composition, Reo Coordination Language, Timed Constraint Automata, Networks of Timed Automata, Model checking, Uppaal.

1 Introduction

In software engineering, a service is an autonomous software entity running on one or more machines and providing a particular functionality to its clients. Service clients are typically other software systems that provide their own services and need the provisioned functionality to fulfil some of their goals. Thus, within one system, various services communicate with each other, e.g., exchange data or collaborate to carry out some activity. Given only limited information about functionalities provided by various services, it is crucial to ensure that these services communicate

[1] Email:natallia.kokash@cwi.nl

[2] Email:jaghoori@cwi.nl

[3] Email:farhad.arbab@cwi.nl

http://dx.doi.org/10.1016/j.entcs.2013.04.004

in a right way to realize a particular business process, i.e., process activities are executed in a right order, each service receives necessary information or accesses a required resource within a right time, etc. The enforcement of communication scenarios that ensure the success of a collaborative service-based process is referred to as service coordination. To increase system verifiability and adaptability, it is desirable to separate actual computation code from coordination code.

Reo [2] is an extensible model for coordination of software components or services wherein complex connectors are constructed out of simple primitives called channels. A channel is a binary relation that defines synchronization and data constraints on its input and output parameters. By composing basic channels, arbitrarily complex interaction protocols can be realized. Previous work shows that most of the behavioral patterns expressible in process modeling notations such as BPMN or UML can be modeled with Reo [5]. A set of tools for automated conversion of such models to Reo have been developed [9]. Each Reo channel has a graphical representation and associated semantics. The most basic semantic model that currently exists for Reo relies on constraint automata [7]. Action constraint automata [15] is a model that generalizes constraint automata by allowing more refined observations on connector ports. In particular, this model can be used to show a data transfer through a composite synchronous region of a Reo network which in basic constraint automata is represented by a single automaton transition. This is needed, e.g., to compute end-to-end time delays in a circuit given communication delays in each channel. In timed Reo [3], special timed channels are introduced to model functional aspects of service coordination protocols such as timeouts or data processing delays.

It is not a trivial task to create a connector that implements a certain behavioral protocol. There are several tools that can help connector designers to detect possible errors in their models. One of them is the animation engine [4]. This tool shows flash animated simulation of designed connectors. For more complex connectors, their formal verification can be performed with the help of simulation and model-checking tools [16] integrated with the Extensible Coordination Tools (ECT), an environment for design and analysis of Reo models. In our previous work [17], we mapped timed Reo to the process algebra mCRL2 which provides a special operator to define relative time constraints on the occurrences of process actions. However, the verification abilities of the mCRL2 toolset with respect to time properties are currently very limited. In particular, a model checker dealing with real time, time-aware simulation facilities, and system property specification language supporting time constraints are not available. Therefore, we need a more powerful tool for analyzing timed workflow and service composition models designed with Reo.

In this paper, we aim to eliminate this gap by integrating ECT with Uppaal [8], a powerful and widely used toolset for real-time system modeling, validation and verification. In Uppaal, distributed real-time systems are modeled as networks of communicating timed automata. Research has demonstrated that rigorous modeling of the behavior of concurrent and distributed systems can prove to be very successful in uncovering design flaws. However, the need to model a system at

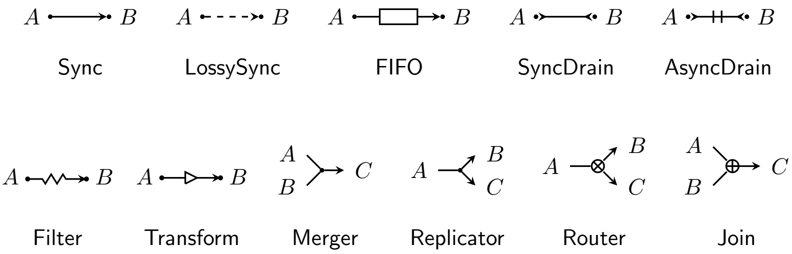


Fig. 1. Graphical representation of basic Reo channels and nodes

the level of automata or process algebras remains an obstacle for most practitioners without expertise in formal methods. They find it more intuitive to describe component or service interactions at a system level [18]. Reo provides a simple and yet powerful formalism for service-based system specification. It is suitable for both scenario-based [6] and workflow-like modeling [5] and together with fully automated translation of graphical models to lower-level formalisms understandable by model checking tools can become an excellent tool for rigorous system design. Here we present a mapping of Reo networks to the Uppaal networks of timed automata. This mapping preserves the compositionality of Reo, i.e., each Reo channel is mapped separately and the behavior of the entire connector can be obtained as a combination of timed automata for channels constituting the connector.

The remainder of this paper is organized as follows. In Section 2, we explain the basics of Reo. In Section 3, we describe the Uppaal networks of extended timed automata. In Section 4, we explain how we model Reo channels with Uppaal timed automata templates. In Section 5, we illustrate the use of Uppaal to analyze a sample Reo workflow model. Finally, in Section 6, we conclude the paper and outline our future work.

2 The Reo Coordination Language

Reo is a coordination language in which components and services are coordinated exogenously by channel-based connectors [2]. Connectors are essentially graphs where the edges are user-defined communication channels and the nodes implement a fixed routing policy. Channels in Reo are entities that have exactly two ends, also referred to as ports, which can be either source or sink ends. Source ends accept data into, and sink ends dispense data out of their channels. Although channels can be defined by users, a set of basic Reo channels (see Figure 1) with predefined behavior suffices to implement rather complex coordination protocols. Among these channels are (i) the Sync channel, which is a directed channel that accepts a data item through its source end if it can instantly dispense it through its sink end; (ii) the LossySync channel, which always accepts a data item through its source end and tries to instantly dispense it through its sink end. If this is not possible, the data item is lost; (iii) the SyncDrain channel has two source ends through which it accepts data simultaneously and loses them subsequently; (iv) the AsyncDrain channel, which accepts data items only through one of its two source channel ends at a time and loses them; and (v) the FIFO channel, which is an asynchronous channel with a buffer of capacity one. Additionally, there are channels for data manipulation. For instance, the Filter channel always accepts a data item at its source end and synchronously passes or loses it depending on whether or not the data item matches a certain predefined pattern or data constraint. Finally, the Transform channel applies a user-defined function to the data item received at its source end and synchronously yields the result at its sink end.

Channels can be joined together using nodes. A node can be a source, a sink or a mixed node, depending on whether all of its coinciding channel ends are source ends, sink ends or a combination of both. Source and sink nodes together form the boundary nodes of a connector, allowing interaction with its environment. Source nodes act as synchronous replicators, and sink nodes as non-deterministic mergers. A mixed node combines these two behaviors by atomically consuming a data item from one of its sink ends at the same time and replicating it to all of its source ends. Additionally, we introduce two special nodes as syntactic sugar in the graphical representation of Reo connectors which are frequently used for dataflow modeling [16]. One of them, Router node, represents a shorthand notation for a Reo circuit behaving as an exclusive router. Another one, Join node, represents a shorthand notation for a component that synchronizes all ends of its incoming channels, forms a tuple of data items received through them and replicates it to the source ends of all its outgoing channels.

The basic set of Reo channels can be extended to enable modeling of specific features of service communication. In particular, timed Reo [3] was introduced to specify time-dependent interaction protocols. A deadline t for the availability of some data can be represented using a channel with a FIFO buffer that loses its data item after t units of time. Another representative example is a *timer channel* (denoted as •—○↠) that can be seen as an asynchronous blocking channel with internal states: when the timer is switched off, the channel consumes any data value, starts the timer and generates a special 'timeout' value at its sink end after a predefined amount of time. Often it is useful to influence the behavior of a timed channel. To enable such control, we define channels that react in a special way to specific data inputs. For example, a so-called *t-timer with off and reset option* allows the timer to be stopped before the expiration of its delay when a special 'off' value is consumed through its source end. Similarly, the 'reset' option allows the timer to be reset to 0 when a special 'reset' value is consumed.

2.1 Semantic models for Reo

The informal description of channel behavior presented above is not sufficient to fully understand the semantics of a Reo connector. The most basic model expressing formally the semantics of Reo is constraint automata [7]. Transitions in a constraint automaton are labeled with sets of ports that fire synchronously, as well as with data constraints on these ports. The constraint automata-based semantics for Reo is compositional, meaning that the behavior of a complex Reo circuit can be obtained from the semantics of its constituent parts using the product operator.

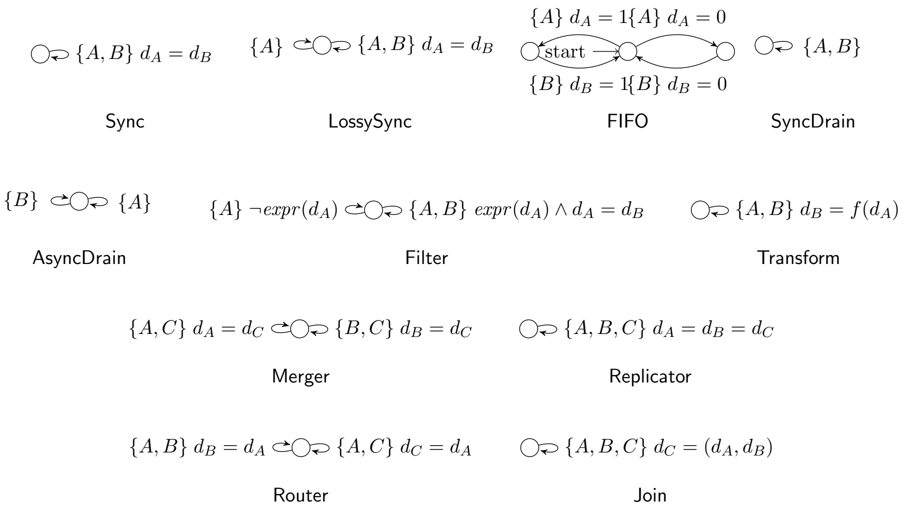


Fig. 2. Constraint automata for basic Reo channels and nodes

Furthermore, the hiding operator can be used to abstract from unnecessary details such as dataflow on the internal ports of a connector.

Definition 2.1 [Constraint automaton (CA)] A constraint automaton $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0)$ consists of a set of states (also called locations) S, a set of port names $\mathcal{N}$, a transition relation $\rightarrow \subseteq S \times 2^{\mathcal{N}} \times DC \times S$, where DC is the set of data constraints over a finite data domain $Data$, and an initial state $s_0 \in S$.

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \rightarrow$. Figure 2 shows the constraint automata for the basic Reo channels. The behavior of any Reo circuit composed of these channels can be obtained by computing the product of their corresponding automata.

Timed constrained automata (TCA) [3] represent constraint automata with clock assignments and timing constraints. They are used to model elements of time-dependent interaction protocols such as timeouts. More formally, TCA can be defined as follows. Let $\mathcal{C}$ be a finite set of clocks. A clock assignment is a function $v : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$. A clock constraint (denoted cc) for $\mathcal{C}$ is a conjunction of atoms of the form $x \bowtie n$ where $x \in \mathcal{C}, \bowtie \in \{<, \leq, >, \geq, =\}$ and $n \in \mathbb{N}$. CC denotes the set of all clock constraints for the set of clocks $\mathcal{C}$.

Definition 2.2 [Timed constraint automaton (TCA) [3]] A TCA is an extended constraint automaton $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0, \mathcal{C}, ic)$ with transition relation $\rightarrow \subseteq S \times 2^{\mathcal{N}} \times DC \times CC \times 2^{\mathcal{C}} \times S$ such that $\mathcal{C}$ is a finite set of clocks and $ic : S \rightarrow CC$ is a function that assigns a clock constraint, called an invariance condition $ic(s)$ to each location s of $\mathcal{A}$.

The definition of a timed constraint automaton is similar to the definition of a standard timed automaton [1]. However, in contrast to the usual timed automata, TCA contain three transition labels: (i) synchronization constraints that represent the set of ports where dataflow is observed simultaneously, (ii) data constraints that

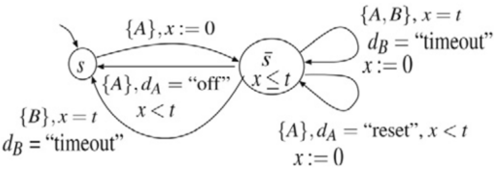


Fig. 3. Timed constraint automaton for the timer channel with off and reset options

enable these transitions and, finally, (iii) clock constraints.

A TCA for a *timer with off and reset options* channel is shown in Figure 3. In this model, the state s represents a timer that is switched off, while the state $\overline{s}$ corresponds to the timer being switched on.

Similarly to constraint automata, product and hiding operators are defined for TCA to obtain the semantics of a timed Reo connector out of the TCA for its basic channels. Since the hiding operator is not essential for understanding the semantics of timed Reo in this paper, we define only the product operator:

Definition 2.3 [Product of TCA [3]] Given two TCA $T_1 = (S_1, \mathcal{N}_1, \rightarrow_1, S_{0,1}, \mathcal{C}_1, ic_1)$ and $T_2 = (S_2, \mathcal{N}_2, \rightarrow_2, S_{0,2}, \mathcal{C}_2, ic_2)$ with disjoint clock sets, the product $T_1 \bowtie T_2$ is defined as an TCA with the location space $S = S_1 \times S_2$, the set $S_0 = S_{0,1} \times S_{0,2}$ of initial locations, the node-set $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$, and the clock set $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$. The location invariance is given by $ic(\langle s_1, s_2 \rangle) = ic_1(s_1) \wedge ic(s_2)$. The edge relation $\rightarrow$ is obtained through the following rules:

$$\frac{s_1 \xrightarrow{N_1, dc_1, cc_1, C_1}_1 s_1', \quad s_2 \xrightarrow{N_2, dc_2, cc_2, C_2}_1 s_2', \quad N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1,\ N_1 \neq \emptyset,\ N_2 \neq \emptyset,\ dc_1 \wedge dc_2 \neq false}{\langle s_1, s_2 \rangle \xrightarrow{N_1 \cup N_2, dc_1 \wedge dc_2, cc_1 \wedge cc_2, C_1 \cup C_2} \langle s_1', s_2' \rangle}.$$

and

$$\frac{s_1 \xrightarrow{N_1, dc_1, cc_1, C_1}_1 s_1', \quad N_1 \cap \mathcal{N}_2 = \emptyset}{\langle s_1, s_2 \rangle \xrightarrow{N_1, dc_1, cc_1, C_1} \langle s_1', s_2 \rangle}, \qquad \frac{s_2 \xrightarrow{N_2, dc_2, cc_2, C_2}_1 s_2', \quad N_2 \cap \mathcal{N}_1 = \emptyset}{\langle s_1, s_2 \rangle \xrightarrow{N_2, dc_2, cc_2, C_2} \langle s_1, s_2' \rangle}.$$

The first rule concerns the "synchronization case" where two edges with common nodes are combined as well as the case where two edges with non-empty "local" node-sets are taken simultaneously. The second and the third rules apply to edges all of whose involved nodes are local to only one of the automata. For the detailed description and semantics of timed automata and TCA refer to [14].

3 Networks of Timed Automata

Timed automata [1] use a dense-time model where a clock variable evaluates to a real number and all clocks progress synchronously. Suppose $\mathcal{B}(C)$ is the set of all clock constraints on the set of clocks C.

Definition 3.1 [Timed Automata] A timed automaton over actions Act and clocks C is a tuple $(L, l_0, \longrightarrow, I)$ representing

- a finite set of locations L (including an initial location l_0);
- the set of edges $\longrightarrow \subseteq L \times \mathcal{B}(C) \times Act \times 2^C \times L$; and,
- a function $I : L \mapsto \mathcal{B}(C)$ assigning an invariant to each location.

An edge (l, g, a, r, l') implies that action 'a' may change the location l to l' by resetting the clocks in r, if the clock constraints in g (as well as the invariant of l') hold. Since we use Uppaal [8], we allow defining variables of type boolean and bounded integers. Variables can appear in guards and updates. The semantics of timed automata changes such that each state will include the current values of the variables as well, i.e., (l, u, v) with v a variable assignment. An action transition $(l, u, v) \xrightarrow{a} (l', u', v')$ additionally requires v and v' to be considered in the corresponding guard and update.

A system may be described as a *network* of communicating timed automata. Semantically, the system can delay if all automata can delay, and can perform an action if one of the automata can perform an internal action or if two automata can synchronize. Synchronization in Uppaal takes place via channels. A binary synchronization channel c in Uppaal can be declared as `chan` c. An automaton edge labeled with $c!$ synchronizes with another edge labeled $c?$. A broadcast channel c is declared as `broadcast chan` c and is used for multi-party communication: one emitter $c!$ can synchronize with an arbitrary number of receivers $c?$ and all channels that can synchronize at a current state must do so. If there are no receivers, the emitter can still execute the $c!$ action, i.e. broadcast sending is never blocking. In a network of timed automata, variables can be defined locally for one automaton, globally (shared between all automata), or as parameters to the automata.

A location can be marked *urgent* in an automaton to indicate that the automaton cannot spend any time in that location. This is equivalent to resetting a fresh clock x in all of its incoming edges and adding an invariant $x \leq 0$ to the location. In a network of timed automata, the enabled transitions from an urgent location may be interleaved with those from other automata (while time is frozen). Like urgent locations, *committed* locations freeze time; moreover, if a process is in a committed location, the next step must involve an edge from one of the committed locations.

Definition 3.2 [Uppaal Model] An Uppaal model consists of: (1) a set of timed automata templates (TAT); (2) global declarations; and, (3) system declarations.

An automata template in an Uppaal model consists of a name, a set of arguments, local declarations and a timed automaton definition (as above); formally, $TAT = (tName, Args, local, Auto)$. Global and local declarations contain the definition of clocks and variables. The network of timed automata to be analyzed is defined in the system declarations by instantiating the timed automata templates.

4 Mapping Reo Channels to Timed Automata

In this section, we show how to translate Reo network specifications to Uppaal networks of timed automata.

Note that despite the same name, Reo channels differ from UPPAAL channels. An UPPAAL channel synchronizes its emitter with its receiver, which is in fact similar to the behavior of a Reo node, i.e., synchronizing the incoming sink channel end with the outgoing source channel end. On the contrary, Reo channels have different behaviors. In our translation, UPPAAL channels refer to Reo nodes, while Reo channels correspond to UPPAAL timed automata templates. The UPPAAL model corresponding to a given Reo network consists of three parts:

Global declarations [Reo nodes]

To model a Reo network, we define a set of UPPAAL channel variables $\mathsf{chan}\, a, b, c\ldots$ with names corresponding to Reo nodes. Besides synchronization, a node needs to pass on the data. As explained in [8], UPPAAL timed automata can exchange data through shared variables: an automaton A_1 assigns a value to a global variable *var* while firing an edge labeled with $c!$ and an automaton A_2 can read this variable at the moment of the synchronization of $c!$ with $c?$ and copy it to some local variable. As UPPAAL supports only bounded integer variables, any data item passing through a Reo circuit must be mapped to integer values. This mapping does not affect the expressiveness of the model for countable data domains.

Automata templates [Reo channel types]

We represent each Reo channel type as a template in UPPAAL where the template parameters are Reo ports. Hence, for mapping all basic Reo channels shown in Figure 1, the following automata templates are created:

$$\begin{array}{ll}
\mathsf{Sync}(\mathsf{chan}\,\&in, \mathsf{chan}\,\&out); & \mathsf{LossySync}(\mathsf{chan}\,\&in, \mathsf{chan}\,\&out);\\
\mathsf{SyncDrain}(\mathsf{chan}\,\&in_1, \mathsf{chan}\,\&in_2); & \mathsf{AsyncDrain}(\mathsf{chan}\,\&in_1, \mathsf{chan}\,\&in_2);\\
\mathsf{Filter}(\mathsf{chan}\,\&in, \mathsf{chan}\,\&out) & \mathsf{Transform}(\mathsf{chan}\,\&in, \mathsf{chan}\,\&out)\\
\multicolumn{2}{c}{\mathsf{FIFO}(\mathsf{chan}\,\&in, \mathsf{chan}\,\&out) \qquad\qquad ;}
\end{array}$$

To define specific expressions or transformation functions in Filter and Transform channels, we introduce a separate template for each distinct version of these channels. In many cases the representation of the transformation functions used in Transform channels will not be possible as neither a general definition mechanism (e.g., λ calculus), nor a library of common mathematical functions is supported in UPPAAL.

Several unrelated transitions with data exchange can be observed at each step in a Reo circuit. Hence, the use of a single global variable for value passing among channels can cause confusion. Instead, we have to parameterize a timed automaton for each Reo channel with the references to global variables this automaton must read and write to. Thus, in data-aware Reo, the template signatures for channels that pass data, i.e., all directed channels FIFO, Sync, LossySync, Filter and Transform, change to:

$$\mathsf{ChannelName}(\texttt{chan}\,\&in,\ \texttt{chan}\,\&out,\ \texttt{int}\,\&d\_in,\ \texttt{int}\,\&d\_out),$$

where d_in and d_out are references to the variables for data exchange on nodes in and out.

Note that no data flow is possible in Reo circuits without an environment that provides and consumes some data tokens to/from the circuit. Such an environment in Reo is defined by external components or services. We refer to the most simple of those components that have exactly one external port, either input or output, and either consume or produce a single data item at a time, as to *readers* and *writers*:

$$\mathsf{Writer}(\texttt{chan}\,\&in);\quad \mathsf{Reader}(\texttt{chan}\,\&out);$$

System definition

To create a Reo network, we instantiate a template for each Reo channel with the variables representing its adjacent nodes. Thus, to create a so called `lossyFIFO` circuit with a source node a, a mixed node b and a sink node c, we declare two automata $lossy_{ab} = \mathsf{LossySync}(a,\, b)$ and $fifo_{bc} = \mathsf{FIFO}(b,\, c)$ corresponding to the circuit constituent channels and execute them in parallel.To feed the aforementioned circuit with data items, we define a *reader* and a *writer*, instantiate them with the boundary nodes of our circuit, and add to the system:

$$\textbf{system}\ writer_a,\ lossy_{ab},\ fifo_{bc},\ reader_c.$$

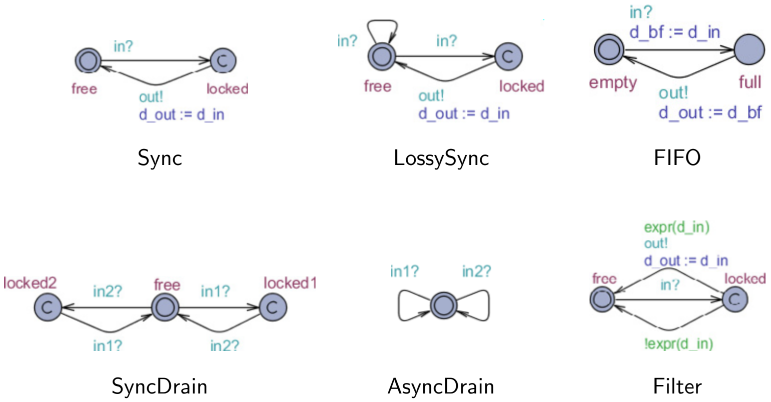
To create a data-aware Reo network, we declare global variables $\texttt{int}\, d\_a,\ d\_b,\ d\_c...$ and pass them as parameters to Reo components and channels:

$$writer_a = \mathsf{Writer}(a,\, d\_a); \qquad lossy_{ab} = \mathsf{LossySync}(a,\, b,\, d\_a,\, d\_b);$$
$$fifo_{bc} = \mathsf{FIFO}(b,\, c,\, d\_b,\, d\_c); \quad reader_c = \mathsf{Reader}(c,\, d\_c);$$

4.1 Timed Automata Templates for Reo Channels

Having in mind the aforementioned mapping concepts, we can define the Uppaal timed automata representing basic Reo channels as shown in Figure 4. All synchronous channels reassign the value of the variable d_in to the value of the variable d_out. The FIFO channel copies the input value d_in to its local variable d_bf which refers to the data value stored in its buffer, and later on assigns this value to the global output variable d_out.

Since in Uppaal timed automata it is not permitted to assign two or more labels on a single automaton edge, we have to introduce an additional state for each synchronous channel. This state, *locked*, is labeled as *committed* (C), which means that the system cannot stay in it and thus triggers an outgoing transition. One



Sync LossySync FIFO

SyncDrain AsyncDrain Filter

Fig. 4. Uppaal timed automata for basic channel types

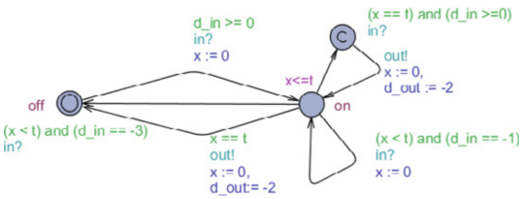


Fig. 5. Uppaal timed automaton for a t-timer channel with *off* and *reset* options

of the drawbacks of this approach is that the UPPAAL simulation engine does not recognize such channels as synchronous and shows them on two different levels of sequence diagrams used for the visualization of the system execution traces.

Timer channels

Figure 5 shows an UPPAAL timed automaton for a *t-timer with off and reset options* channel. Here we assume that circuits operate with any integer value $d\_in \geq 0$ while $d\_in = -1$, $d\_in = -2$ and $d\_in = -3$ are used to represent values corresponding to the 'reset', 'timeout' and 'off' signals, respectively. Note that Arbab et al. [3] abstract from data constraints and the TCA presented in this paper do not show guards that ensure that the automaton reacts on these special inputs in a special way. Instead in our model, any data input $d\_in \geq 0$ switches on the timer represented by a local clock x and the automaton goes from the state *off* to the state *on*. It stays in this state for a predefined amount of time, i.e., $x \leq t$. Before this time expires, the timer can be reset back to 0 by a data input $d\_in = -1$. At time $x = t$, the automaton returns to the initial state and generates a special output value $d\_out = -2$. If another data input $d\_in \geq 0$ is available at time $x = t$, the timer can generate an output value, be switched off and immediately turned on with its clock reset to 0. This is modeled as a transition to an intermediate committed location and then back to the *on* location without any time delay.

Note that other variants of timer channels can be defined in Reo. For example, one of the useful modifications of the aforementioned timer channel would be a delaying FIFO that instead of generating the 'timeout' signal would dispense the data input it received earlier through its source end. We will use such a channel in the next section to model an activity with a known processing time.

4.2 Implementing Reo join operator in Uppaal

Due to the fact that UPPAAL channels synchronize non-deterministically if several combinations of input and output edges with the same name are enabled, we can model the behavior of Reo Merger and Router nodes by instantiating all joint channels with a common port name. However, to reproduce the behavior of Replicator nodes, we need to involve broadcast channels. Since UPPAAL specification language does not support polymorphism or channel type inheritance, we cannot pass broadcast channels as parameters to the aforementioned templates. Hence, we need to create separate templates for all basic Reo channels connected to Replicator nodes. For every channel type, e.g., FIFO, we introduce two additional templates with the following signatures:

$$\texttt{FIFO\_r}(\texttt{chan}\,\&in, \texttt{broadcast chan}\,\&out,\ \texttt{int}\,out\_m,\ \texttt{int}\,\&out\_n);$$

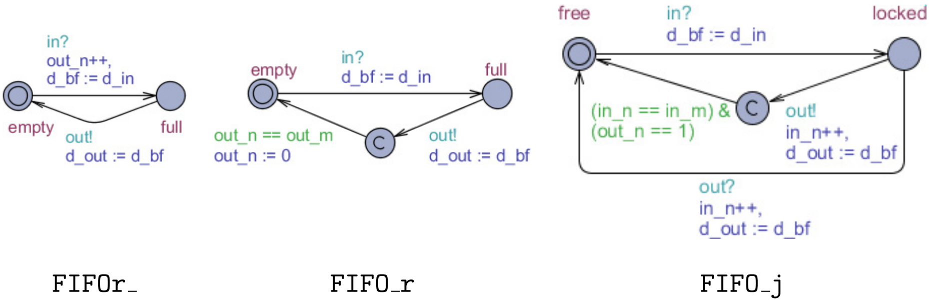
$$\texttt{FIFOr\_}(\texttt{broadcast chan}\,\&in, \texttt{chan}\,\&out,\ \texttt{int}\,\&out\_n);$$

Here we use a suffix r after the Reo channel name to refer to a Replicator node. For example, the template `FIFO_r` represents a FIFO with a sink end connected to a Replicator node, and `FIFOr_` refers to a FIFO with a source end connected to a Replicator node. Apart from that, we need to guarantee that an edge corresponding to a data item dispensed out of a `FIFO_r` channel fires only if all outgoing Reo channels are ready to accept this data item. To implement such a behavior in UPPAAL, each automaton has to know how many connected channels it must synchronize with. Therefore we provide each Reo channel with this information using an integer variable out_m that refers to the number of outgoing channels from a Replicator node. Automata corresponding to channels with source ends connected to the Replicator (e.g., `FIFO_r`) increase a shared variable out_n while an automaton corresponding to a channel with the sink end connected to the Replicator, (e.g., `FIFOr_`) checks that it synchronizes with the correct number of outgoing channels defined by its parameter out_m, i.e., $out\_m = out\_m$. For example, to create a circuit with a source node a, an internal node b and sink nodes c and d, which consumes a data item into one FIFO channel and replicates it into two other FIFO channels, we declare global channel variables `chan` a, c, d and a broadcast channel variable `broadcast chan` b together with three instances of the FIFO templates synchronized on a broadcasting channel b:

$$fifo_{ab} = \texttt{FIFO\_r}(a,\ b,\ 2,\ out\_b); \quad fifo_{bc} = \texttt{FIFOr\_}(b,\ c,\ out\_b);$$
$$fifo_{bd} = \texttt{FIFOr\_}(b,\ d,\ out\_b);$$

where `int` out_b is a variable used to count how many automata edges labeled with $b?$ must synchronize with $b!$

Figure 6 shows the UPPAAL implementation of FIFO channels connected to



FIFOr_ FIFO_r FIFO_j

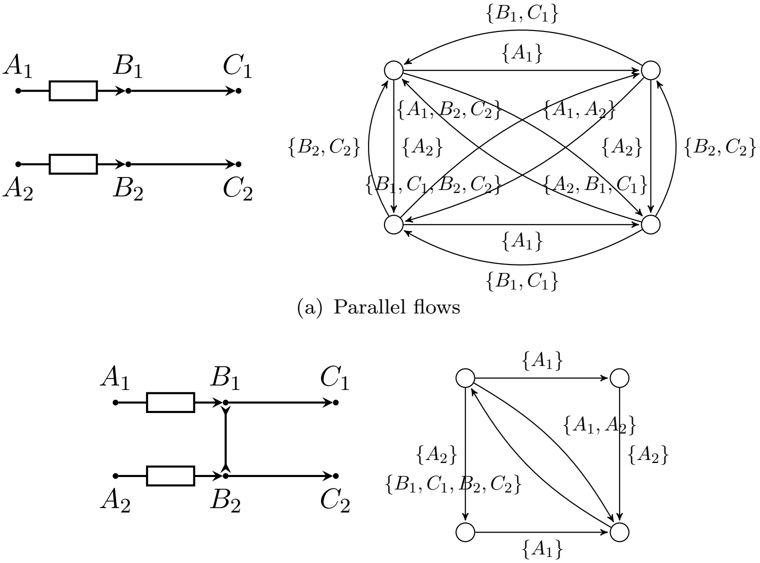
Fig. 6. Modeling the behavior of channels connected to the `Replicate` and `Join` nodes

Replicator and Join nodes. For any broadcasting channel c in Uppaal, an automaton edge $c!$ synchronizes with all enabled edges $c?$ in this step. However, if several edges $c!$ can be triggered at some step, only one of them will synchronize with edges $c?$. Taking this into account, the behavior of the Join node can be modeled as follows: a first Reo channel with its sink end connected to the Join node that receives a data input triggers a sending $c!$ edge, while all subsequent incoming channels have to synchronize with it via $c?$ edge and increase the shared variable in_n used to count the number of synchronized processes. Before completing the transition, the Reo channel that triggered $c!$ has to check that all Reo channels connected to the Join node are ready to provide their inputs, i.e., $in\_n == in\_m$ and the outgoing channel is ready to accept the synchronized input, i.e., $out\_n == 1$. The template for a Reo channel, e.g., FIFO, with the sync end connected to the Join node looks as follows:

$$\texttt{FIFO\_j}(\texttt{chan}\,\&in, \texttt{broadcast chan}\,\&out, \texttt{int}\,in\_m, \texttt{int}\,\&in\_n, \&out\_n,$$
$$\texttt{int}\,\&d\_in, \texttt{int}\,\&d\_out);$$

Channels with their source ends connected to Join nodes behave similarly to the channels connected to Replicate nodes. However, if Join nodes are used in a Reo circuit, they affect the global data structure used to describe the data elements exchanged through channels in this circuit. A Join node synchronizes all incoming edges and forms a tuple from all received data inputs. To reproduce such a behavior in Uppaal, we need to use an array of integer values $\texttt{int}\,d\_in[N]$ to store all data inputs. Consequently, global variables $d\_x[N]$ are needed to further propagate this data along the circuit. Thus, for all Reo channels following some Join node, the arrays of integer variables are used for data passing. Ideally, we would use arrays of dimensions $N = in\_m$, but the Uppaal specification language requires N to be a predefined integer constant. We can define N as the maximal sum of the arities of Join nodes on all execution paths in a Reo network. Templates for channels with their source and sink ends connected to Join and Replicator nodes are formed as a combination of the templates presented above.

One of the fundamental differences between Reo and Uppaal with respect to the concurrency modeling is that the automata synchronization in Uppaal is possible only through communicating channels while in Reo synchronous data flow can be



(a) Parallel flows

(b) Barrier synchronization

Fig. 7. Parallel flows and synchronization in Reo

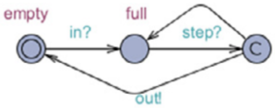


Fig. 8. FIFO with global synchronization

observed on various ports without their interaction. Consider a network consisting of two chains of FIFO channels shown in Figure 7(a). According to its (T)CA semantics, if both buffers are full, the simultaneous data flow on ports B_1 and B_2 can be observed. However, our Uppaal mapping will show only an interleaving of actions corresponding to the data flow on ports B_1 and B_2. Such treatment of "accidental" synchronization will lead to the deadlock in the Uppaal representation barrier synchronization template shown in Figure 7(b). To force Uppaal to consider all combinations of synchronous events in Reo, we can introduce a simple automaton with one state and one loop transition representing a global "next step" event using an emitter channel *step*!. Reo entities that provide data, such as buffered channels, writers and external components, can synchronize with this event and afterwards decide non-deterministically whether to release data or not. Figure 8 shows a representation of a FIFO channel synchronizing with such a global event through the receiver channel *step*?.

The relation between the TCA semantics for Reo circuits and their representation using Uppaal networks of timed automata can be expressed by the following proposition:

Proposition 4.1 *Given a TCA $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0, \mathcal{C}, ic)$ and a corresponding network of communicating timed automata $NTA = \{TAT_i \,|\, i = 1..n\}$, a transition*

$s \xrightarrow{N,dc,cc,C} s'$ *exists iff for any* $TAT_i = (L_i, l_{0,i}, \longrightarrow_i, I_i) \in NTA$ *there exists an execution path* $l, l_1, ...l_m, l', \mid l \in L_i,\ l' \in L_i,\ \forall j = 1..m,\ l_j \in L_i$, *where* l *and* l' *are not marked and* $l_1, ..., l_m$ *are masked as* committed *locations.*

The intuition behind this statement is as follows: by construction, for every location in a TCA for a basic Reo channel, we introduce a non-committed location in the corresponding timed automaton, while all committed locations in the instantiated templates represent data flow through Reo ports shown as TCA transition labels. Thus, we can define a weak bisimulation relation between TCA and the constructed networks of timed automata. The goal then is to show that the introduced Uppaal synchronization templates preserve the semantics of the TCA product operation. Due to the lack of space we cannot provide a formal proof here.

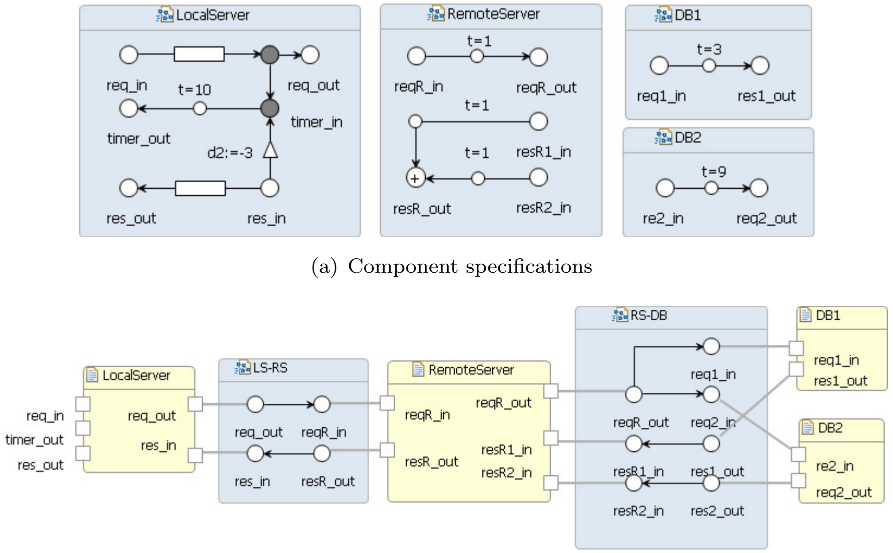
5 Example: Remote Distributed Data Request

In this section, we consider an example of a Reo circuit modeling a remote distributed data request and show how this model is mapped to Uppaal networks of timed automata.

Our scenario is as follows. A *local server* receives a data request from its user. It switches on a local clock to keep track of the request processing time and forwards it to a *remote server*. The *remote server* replicates the request and sends it to two *databases*. After processing the requests, the databases return retrieved data to the remote server. The remote server combines the data received from both databases and returns the result to the local server. If the request processing has been carried out within the time limit, the *local server* returns the result to the user, otherwise, a timeout exception is generated.

A Reo model for this scenario is shown in Figure 9. We modeled the behavior of the *local server*, *remote server* and two *databases* using four Reo connectors as shown in Figure 9(a). The *local server* accepts a user request through its internal buffer and in the next step forwards this request to the *remote server* and starts the internal clock. The clock is modeled using a *t-timer with off and reset options*. When a response from the *remote server* is received, the signal passes through a Transform channel that generates an 'off' option to switch off the timer. The response is returned to the user through the port *res_out*. The *remote server* replicates the request through a Replicate node and forwards it to the two *databases*. Later on, it accepts responses from the *databases* and joins them using a Join node *resR_out*. All basic activities are modeled using *t-timer* channels that behave exactly like FIFO channels with internal clocks, i.e., accept data items at their source ends and after some time dispense them through their sink ends. The behavior of *databases* that accept data requests and provide the results is also modeled using such *t-timer* channels. Figure 9(b) shows an integrated Reo model for this scenario where the internal details of each component are hidden.

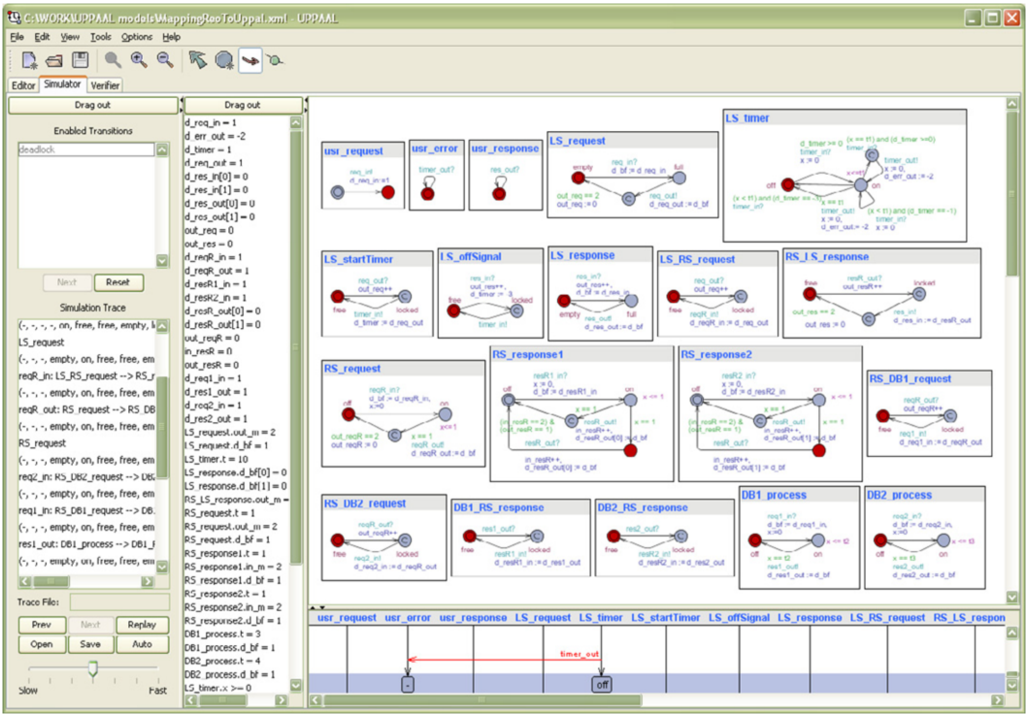
The generated Uppaal specification for the aforementioned scenario is shown in Figure 11. We define two Uppaal global variables for each Reo node: one of the



(a) Component specifications

(b) Integrated model

Fig. 9. Reo model of the Remote Distributed Data Request scenario



```
(-, -, -, -, on, free, free, empty, l
LS_request
(-, -, -, empty, on, free, free, em
reqR_in: LS_RS_request --> RS_r
(-, -, -, empty, on, free, free, em
reqR_out: RS_request --> RS_DB
(-, -, -, empty, on, free, free, em
RS_request
(-, -, -, empty, on, free, free, em
req2_in: RS_DB2_request --> DB2
(-, -, -, empty, on, free, free, em
req1_in: RS_DB1_request --> DB1
(-, -, -, empty, on, free, free, em
res1_out: DB1_process --> DB1_F
(-, -, -, empty, on, free, free, em
```

```
d_req_in = 1
d_err_out = -2
d_timer = 1
d_req_out = 1
d_res_in[0] = 0
d_res_in[1] = 0
d_res_out[0] = 0
d_res_out[1] = 0
out_req = 0
out_res = 0
d_reqR_in = 1
d_reqR_out = 1
d_resR1_in = 1
d_resR2_in = 1
d_resR_out[0] = 0
d_resR_out[1] = 0
out_reqR = 0
in_resR = 0
out_resR = 0
d_req1_in = 1
d_res1_out = 1
d_req2_in = 1
d_res2_out = 1
LS_request.out_m = 2
LS_request.d_bf = 1
LS_timer.t = 10
LS_response.d_bf[0] = 0
LS_response.d_bf[1] = 0
RS_LS_response.out_m =
RS_request.t = 1
RS_request.out_m = 2
RS_request.d_bf = 1
RS_response1.t = 1
RS_response1.in_m = 2
RS_response1.d_bf = 1
RS_response2.t = 1
RS_response2.in_m = 2
RS_response2.d_bf = 1
DB1_process.t = 3
DB1_process.d_bf = 1
DB2_process.t = 4
DB2_process.d_bf = 1
LS_timer.x >= 0
```

Fig. 10. Remote Distributed Data Request scenario simulation

type chan to join Reo channel ends, and one of the type int for data value passing. Additionally, for all Replicate and Join nodes that are declared as broadcast channels, i.e., *req_out*, *res_in*, *reqR_out* and *resR_out*, we define variables *out_req*, *out_res*, *out_reqR* and *out_resR* to count the number of synchronized broadcast channels.

All variables for the nodes that follow the Join node *resR_out* are arrays of two elements, i.e., `int` $d\_resR\_out[2]$, $d\_res\_in[2]$, $d\_res\_out[2]$. All automata declarations are based on automata templates discussed in Section 4.

Figure 10 shows a screenshot of the remote distributed data request scenario modeled in UPPAAL. By changing time delays assigned to the *local server* to generate the timeout event and to *databases* to model time delay for request processing, one can observe the system exhibit different interesting behavior. For instance, when the process request cannot be handled within a predefined timeout $t_1 = 10$, the system will get into a deadlock: the *t-timer with off and reset options* will generate a 'timeout' signal and expect a data item $d\_in \geq 0$ to restart the timer. Consequently, the Transform channel will not be able to dispense a data item, and, thus, it will not accept an input from the synchronous channel Syncjr(*resR_out*, *res_in*,...) represented by the automaton *RS_LS_response*. This automaton will not be able to leave its committed location and the system will end up in a deadlock. The absence of deadlocks in a Reo model as well as other system properties can be checked automatically by the UPPAAL model checker given formulae in a subset of the Computation Tree Logic (CTL). Examples of system properties specified in the UPPAAL query language can be found in [8]. However, note that the presence of deadlocks in UPPAAL simulations of timed automata obtained from Reo circuits using the presented approach does not necessarily mean a conceptual mistake in a circuit design. Deadlocks in UPPAAL simulation of Reo circuits simply correspond to the absence of dataflow through these circuits. For example, the deadlocks in the generated UPPAAL execution traces may appear due to the fact that we can enter a committed location of a timed automaton for a synchronous Reo channel without being sure that all nodes involved in a synchronous transaction are ready to provide or consume data. In this case, we can revert several steps back in the UPPAAL automata simulations and consider other execution traces. The presented translation is more suitable for a guided simulation and reachability analysis of timed dataflow models, rather than their automated verification, due to the potential difficulties to formally describe desired system properties.

6 Conclusions and Future Work

In this paper, we presented an approach for mapping Reo networks to the networks of timed automata to enable their timed analysis with the UPPAAL model checker. Despite many conceptual differences in these two specification formalisms, we were able to compositionally map one into the other preserving the behavior of dataflow models in Reo. The most closely related work to ours is the SAT-based verification of timed component connectors [12]. In this work, an approach for bounded model checking of TCA is proposed by translating systems of TCA into propositional logic with linear arithmetic constraints. Since TCA provide a semantic model for timed Reo, this work can be used for the analysis of Reo circuits. However, our work provides a more straightforward approach and uses the well-established UPPAAL model checker. In [13], a compositional model for real-time coordination in dataflow networks is presented. This approach is interesting for its combination of synchronous and asynchronous communication patterns, but it significantly differs from the principle of extensible channel-based coordination used in Reo. An important issue for our future work is to formally establish the relation between TCA and the semantics of the generated Uppaal networks of timed automata. The intuition behind it is as follows: a transition in a TCA corresponds to a path in the Uppaal network of timed automata that leads from one location without committed states to another location without committed states.

A translation from a subset of constraint automata to timed automata has been discussed in [11] in the context of schedulability analysis of real-time actors coordinated by Reo. In that work, a Reo connector is considered to be locally deployed, i.e., no distributed implementation of Reo, and furthermore, the coordination is assumed to happen in zero time. The effect of communication delay is studied as an orthogonal concern.

Together with previously developed tools for mapping BPMN, BPEL and UML specifications to Reo, our work constitutes an important step in enabling timed analysis of business process models through their seamless transformation to lower-level formalisms for which powerful established tools exist. In particular, our work is directly related to the mapping of workflow notations to Uppaal. For example, Gruhn and Laue [10] provide a set of patterns for mapping a basic workflow specification notation consisting of an activity and four common control structures, AND-split, OR-split, AND-join and OR-join, to the Uppaal timed automata. We find the patterns proposed in this paper disputable as they do not preserve the semantics of the original notation. For example, the proposed mapping of the AND-Split node rather corresponds to a structure that imitates the behavior of a variable: after the value of the variable is set, it can be read arbitrarily many times. However, in this template, there is no guarantee that the AND-split node replicates data to all outgoing branches simultaneously.

Another direction in our future work includes the mapping of action-constraint-automata-based semantics for Reo [15] to Uppaal networks of timed automata to enable tool support for performance analysis of Reo with communication delays. We also believe that this work will help us to develop a suitable "handshaking" protocol for a distributed real-time implementation of Reo.

References

[1] Alur, R. and D. Dill, *A theory of timed automata*, Theoretical Computer Science **126** (1994), pp. 183–235.

[2] Arbab, F., *Reo: A channel-based coordination model for component composition*, Mathematical Structures in Computer Science **14** (2004), pp. 329–366.

[3] Arbab, F., C. Baier, F. de Boer and J. Rutten, *Models and temporal logical specifications for timed component connectors*, Software and Systems Modeling **6** (2007), pp. 59–82.

[4] Arbab, F., C. Koehler, Z. Maraikar, Y. Moon and J. Proenca, *Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools*, Tool demo session at FACS '08 (2008).

[5] Arbab, F., N. Kokash and M. Sun, *Towards using reo for compliance-aware business process modelling*, in: *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'08)*, LNCS **17** (2008), pp. 108–123.

[6] Arbab, F. and M. Sun, *Synthesis of Connectors from Scenario-based Interaction Specifications*, in: *Proceedings of the International Symposium on Component Based Software Engineering (CBSE'08)*, LNCS **5282** (2008).

[7] Baier, C., M. Sirjani, F. Arbab and J. Rutten, *Modeling Component Connectors in Reo by Constraint Automata*, Science of Computer Programming **61** (2006), pp. 75–113.

[8] Behrmann, G., A. David and K. G. Larsen, *A tutorial on UPPAAL*, in: M. Bernardo and F. Corradini, editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, LNCS **3185** (2004), pp. 200–237.

[9] Changizi, B., N. Kokash and F. Arbab, *A unified toolset for business process model formalization*, in: J. Happe and B. Buhnova, editors, *Proc. FESCA 2010* (2010), pp. 147–156.

[10] Gruhn, V. and R. Laue, *Using timed model checking for verifying workflows*, in: F. J. Cordeiro, J., editor, *Proceedings of the International Workshop on Computer Supported Activity Coordination* (2005), pp. 75–88.

[11] Jaghoori, M. M., O. Hlynsson and M. Sirjani, *Networks of real-time actors: Schedulability analysis and coordination*, in: *Proc. Formal Aspects of Component Software (FACS'11)*, 2011, to appear.

[12] Kemper, S., *SAT-based verification for timed component connectors*, ENTCS **255** (2009), pp. 103–118.

[13] Kemper, S., *Compositional construction of real-time dataflow networks*, in: *COORDINATION*, 2010, pp. 92–106.

[14] Kemper, S., "Modelling and Analysis of Real-Time Coordination Patterns," PhD thesis, CWI (2011).

[15] Kokash, N., B. Changizi and F. Arbab, *A semantic model for service composition with coordination time delays*, in: Jin Song Dong and Huibiao Zhu, editors, *Proc. ICFEM*, LNCS 6447, 2010, pp. 106–121.

[16] Kokash, N., C. Krause and E. de Vink, *Data-aware design and verification of service composition with Reo and mCRL2*, in: *Proc. of SAC 2010* (2010), pp. 2406–2413.

[17] Kokash, N., C. Krause and E. de Vink, *Time and data aware analysis of graphical service models in Reo*, in: *Proc. SEFM'10* (2010).

[18] Uchitel, S., R. Chatley, J. Kramer and J. Magee, *System architecture: the context for scenario-based model synthesis*, in: *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering* (2004), pp. 33–42.

```
const int t1 = 10, t2 = 3, t3 = 9;
//Nodes and shared variables
//Local server
chan req_in, timer_out, res_out, timer_in;
broadcast chan req_out, res_in;
int d_req_in = 0, d_err_out = 0, d_timer = 0, d_req_out = 0, d_res_in[2], d_res_out[2];
int out_req = 0, out_res = 0;
//Remote server
chan reqR_in, resR1_in, resR2_in;
broadcast chan reqR_out, resR_out;
int d_reqR_in = 0, d_reqR_out = 0, d_resR1_in = 0, d_resR2_in = 0, d_resR_out[2] = 0;
int out_reqR = 0, in_resR = 0, out_resR = 0;
//Databases
chan req1_in, res1_out, req2_in, res2_out;
int d_req1_in = 0, d_res1_out = 0, d_req2_in = 0, d_res2_out = 0;
//Reo channels
//User
usr_request = Writer(req_in, d_req_in);
usr_error = Reader(timer_out);
usr_response = Reader(res_out);
//Local server
LS_request = FIFO_r(req_in, req_out, 2, out_req, d_req_in, d_req_out);
LS_startTimer = Syncr_(req_out, timer_in, out_req, d_req_out, d_timer);
LS_timer = TimerOffReset(timer_in, timer_out, t1, d_timer, d_err_out);
LS_offSignal = Transformr_(res_in, timer_in, out_res, d_timer);
LS_response = FIFOr_(res_in, res_out, out_res, d_res_in, d_res_out);
//Communication LS-RS
LS_RS_request = Syncr_(req_out, reqR_in, out_req, d_req_out, d_reqR_in);
RS_LS_response = Syncjr(resR_out, res_in, 2, out_resR, out_res, d_resR_out, d_res_in);
//Remote server
RS_request = Timer_r(reqR_in, reqR_out, 1, 2, out_reqR, d_reqR_in, d_reqR_out);
RS_response1 = Timer_j(resR1_in, resR_out, 1, 2, in_resR, out_resR, d_resR1_in, d_resR_out[0]);
RS_response2 = Timer_j(resR2_in, resR_out, 1, 2, in_resR, out_resR, d_resR2_in, d_resR_out[1]);
//Communication RS-DBs
RS_DB1_request = Syncr_(reqR_out, req1_in, out_reqR, d_reqR_out, d_req1_in);
RS_DB2_request = Syncr_(reqR_out, req2_in, out_reqR, d_reqR_out, d_req2_in);
DB1_RS_response = Sync(res1_out, resR1_in, d_res1_out, d_resR1_in);
DB2_RS_response = Sync(res2_out, resR2_in, d_res2_out, d_resR2_in);
//Databases
DB1_process = Timer(req1_in, res1_out, t2, d_req1_in, d_res1_out);
DB2_process = Timer(req2_in, res2_out, t3, d_req2_in, d_res2_out);
system
  usr_request, usr_error, usr_response, LS_request, LS_startTimer,
  LS_timer, LS_offSignal, LS_response, LS_RS_request, RS_LS_response, RS_request,
  RS_response1, RS_response2, RS_DB1_request, RS_DB2_request,
  DB1_RS_response, DB2_RS_response, DB1_process, DB2_process;
```

Fig. 11. Uppaal specification for the Remote Distributed Data Request scenario