Under consideration for publication in Formal Aspects of Computing

Reo + mCRL2: A Framework for Model-Checking Dataflow in Service Compositions

Natallia Kokash[*,1], Christian Krause[**,2] and Erik de Vink[***]

[*]Centrum Wiskunde & Informatica (CWI), [**]Hasso Plattner Institute (HPI), [***]Technische Universiteit Eindhoven (TU/e)

Abstract. The paradigm of service-oriented computing revolutionized the field of software engineering. According to this paradigm, new systems are composed of existing stand-alone services to support complex cross-organizational business processes. Correct communication of these services is not possible without a proper coordination mechanism. The Reo coordination language is a channel-based modeling language that introduces various types of channels and their composition rules. By composing Reo channels, one can specify Reo connectors that realize arbitrary complex behavioral protocols. Several formalisms have been introduced to give semantics to Reo. In their most basic form, they reflect service synchronization and dataflow constraints imposed by connectors. To ensure that the composed system behaves as intended, we need a wide range of automated verification tools to assist service composition designers. In this paper, we present our framework for the verification of Reo using the mCRL2 toolset. We unify our previous work on mapping various semantic models for Reo, namely, constraint automata, timed constraint automata, coloring semantics and the newly developed action constraint automata, to the process algebraic specification language of mCRL2, address the correctness of this mapping, discuss tool support, and present a detailed example that illustrates the use of Reo empowered with mCRL2 for the analysis of dataflow in service-based process models.

Keywords: formal methods for service-oriented computing; model checking; coordination languages

1. Introduction

The paradigms of service-oriented computing and model-driven development have changed the way business process supporting software is developed. With the evolution of service-oriented computing, the focus in software development shifted from applications to the integration of reusable services, which are autonomous, loosely coupled software components that can be assembled into more complex systems via publicly available

Correspondence and offprint requests to: Natallia Kokash, Centrum Wiskunde & Informatica, P.O. Box 94079, 1090 GB Amsterdam, the Netherlands, email natallia.kokash@cwi.nl

[1] Supported by IST COMPAS FP7–ICT–2007–1, contract 215175

[2] Supported by a grant of the research school in 'Service-Oriented Systems Engineering' at the Hasso Plattner Institute (HPI)

interfaces. Concerning the organization of software development process, a model-driven approach for service-oriented systems is commonly embraced. More precisely, business processes are described in high-level task-oriented modeling notations such as the Business Process Modeling Notation (BPMN) and UML activity diagrams with their further stepwise and preferably automated translation to lower-level and more detailed specifications and execution code. In the context of web services, the target specifications are given in the Business Process Execution Language (BPEL).

When modeling a service-based process, the developer focuses on the seamless integration of existing services, rather than on the implementation of new functional modules. This integration or composition of services requires a proper synchronization, buffering and ordering of the exchanged messages, as well as a transformation of data formats of different services. An essential feature of a process modeling notation is a clear definition of its execution semantics, supported by efficient tools for the validation, verification and performance analysis of resulting process models. In the past decades, various modeling notations and tools for workflow and services have been proposed [HOP05, GMRMS07]. They vary at different levels, ranging from the particular graphical syntax for process model definitions to the expressiveness and extensibility of functional and organizational aspects of the model. However, general formalisms for verifiable process design, such as process algebras or finite-state machines, are typically too low level. Therefore, the encoding of high-level modeling notations such as BPMN and UML into these formalisms frequently yields a complex and barely understandable description of the original system. Moreover, the analysis results are hard to trace back into the original model. The abundance of process definition languages on the one hand, and elemental analysis techniques on the other, suggests that an intermediate between the design of a workflow process and its analysis is required. Petri nets [Aal98, Aal99] are a prime example of such a model. Various extensions –in particular featuring reset and inhibitor arcs, color, time and hierarchy– provide solid ground to workflow analysis. However, as Petri nets are token-driven by nature, they are on their own insufficient for dataflow verification. While the routing of messages in a process model can be adequately captured with traditional Petri nets, the data manipulation aspect requires more powerful concepts [HKS+05, TAS08].

Conceptually, service-oriented computing is similar to exogenous coordination, which advocates the separation of computation provided by services on the one hand, and their coordination provided by some 'glue code' [Arb96] on the other. The Reo coordination language [Arb04] adheres to this principle and provides concepts and tools to construct such glue code to integrate services given specifications of their behavioral interfaces. A Reo process model is essentially a set of complex connectors (also called circuits or networks) composed of channels which provide the basic form of interaction between two parties by imposing constraints on the data they exchange. Along with the graphical notation and intuitive meaning of channel behavior, several formal semantic models for Reo have been developed. This makes it possible to analyze the connector behavior automatically using simulation and model checking techniques as well as to generate executable code from graphical models. These semantic models, in particular, constraint automata (CA) [BSAR06], timed constraint automata (TCA) [ABBR07], coloring semantics [CCA07] and action constraint automata (ACA) [KCA10], require the development of special software tools that can, first, automatically build the corresponding model given a graphical Reo circuit and, second, enable the analysis of its functional and non-functional properties. Several tools implemented as Eclipse plug-ins have been developed at the CWI in Amsterdam to support the design and analysis of Reo circuits [AKM+08]. These tools include algorithms to compute constraint automata and coloring tables used to model check and animate the execution of graphical Reo circuits [AKM+08]. Moreover, the Vereofy [BBKK09] tool developed at the University of Dresden provides model checking facilities for verifying properties of constraint automata.

In spite of the above, the tailored development of efficient verification tools requires substantial man power over an extensive period of time. Due to the need to implement sophisticated optimization techniques and to analyze the data structures involved, it may take considerable time to leap before a newly developed tool has matured and has become suited for the verification of large system specifications. Therefore, instead of the development of novel verification tools it may be profitable to apply existing state-of-the-art model checking tools for verifying Reo networks. What we need for Reo is a tool that preserves compositional construction of process models, accept automata with multiple transition labels as input and support full-featured data analysis. `mCRL2` is a verification toolset with a specification language that enables the compositional construction of distributed programs by synchronizing actions of communicating processes [Gea07]. It also supports algebraic data types. Therefore, it ideally suits our goal. Moreover, the toolset provides a tool for state-space generation which is compatible with the input format for another popular verification toolset: CADP. In this paper, we bring together our work on the application of `mCRL2` to the verification of Reo connectors. Given graphical Reo models, we automatically generate equivalent `mCRL2` specifications for their

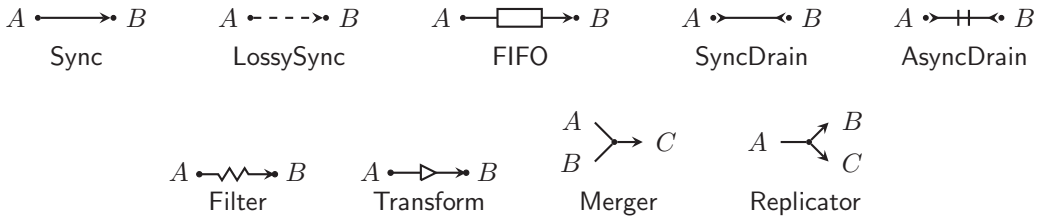


Fig. 1. Graphical representation of basic Reo channels and nodes

subsequent model checking and other analysis with the mCRL2 toolset. Depending on the level of expressivity needed for the verification of a certain property, several kinds of translation are supported:

- mCRL2 description of observable data flow on channel ports for control flow analysis with representation of data constraints imposed by Reo channels as mCRL2 action parameters for data flow analysis [KKV10a].
- Time-aware mapping to enable analysis of timed constraints imposed by Reo channels [KKdV10].
- A translation that relies on coloring semantics for Reo to deal with context-depend channels [KKV10b].
- A mapping based on action constraint automata to enable analysis of dataflow within synchronous regions [KCA10].

In this paper, we focus on constraint automata semantics for Reo. For this base model we discuss our encoding strategy in detail and prove order-independence of our construction and soundness of the translation. Once this is set up, data and context information by colors can be added straightforwardly. Also, the translation to mCRL2 of the extensions of the constraint automaton model, viz. timed constraint automata and action constraint automata, follows the same lines and are presented in brief. More specifically, in Section 2, we introduce the Reo coordination language and some of its semantics. In Section 3, we present the mCRL2 specification language and describe the translation of Reo to mCRL2. In Section 4, we formally prove the correctness of this translation. In Section 5, we illustrate the application of Reo empowered by mCRL2 to the analysis of a service-based process model. The tool support for our framework is addressed in Section 6. Sections 7 and 8 discuss related work and outline conclusions.

2. The Reo Coordination Language

Reo is a coordination language in which components and services are coordinated exogenously by channel-based connectors [Arb04]. 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 channel. 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 the sink end and if this is not possible, the data item is lost; (iii) the SyncDrain channel, which is a channel with two source ends that accept 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 moment in time and loses it; 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 time and replicating it to all of its source ends.

The basic set of Reo channels can be extended to enable modeling of specific features of service communication. For example, probabilistic Reo channels [Bai05] are used to build communication networks with unreliable links, while timed Reo [ABBR07] channels were 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* 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 *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.

The informal description of Reo as given above is rather incomplete. Clarke et al. [CPLA08] argue that the intuition about dataflow networks is insufficient to determine how connectors behave. We refer to the literature for a more detailed exposition, e.g. [Arb04, BSAR06, ABB$^+$05, CCA07, KCA10].

2.1. Constraint Automata

The most basic model expressing formally the semantics of Reo is constraint automata [BSAR06]. 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. 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 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 from these channels can be obtained by computing the synchronous product of the corresponding automata. For example, at the left of Figure 3 we have a small Reo connector consisting of a LossySync, say with ports A and B, and a FIFO, say with ports C and D, that are connected at the respective ports B and C. For simplicity, we use the singleton set $Data = \{x\}$ as data domain. In Section 4 we will address this more precisely, introducing the product operator $\bowtie_\gamma$ with γ a so-called synchronization function that in the case of the LossySync of Figure 3 identifies the ports B and C.

Thus, a Reo connector can be interpreted as the synchronized product of the CAs of all its constituents [BSAR06]. The semantics of the full connector is obtained compositionally building on the standard CAs assigned to the primitive channels.

2.2. Timed Constraint Automata

Timed constrained automata (TCA) [ABBR07] represent constraint automata with clock assignments and timing constraints. They are useful to formally model elements of time-dependent interaction protocols such as timeouts. More formally, a 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 [ABBR07]). A TCA is an extended constraint automaton $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0, ic)$ with transition relation $\rightarrow \subseteq S \times 2^{\mathcal{N}} \times DC \times CC \times 2^{\mathcal{C}} \times S$ such that $dc \in DC(\mathcal{N})$, $\mathcal{C}$ 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 $\mathcal{A}$.

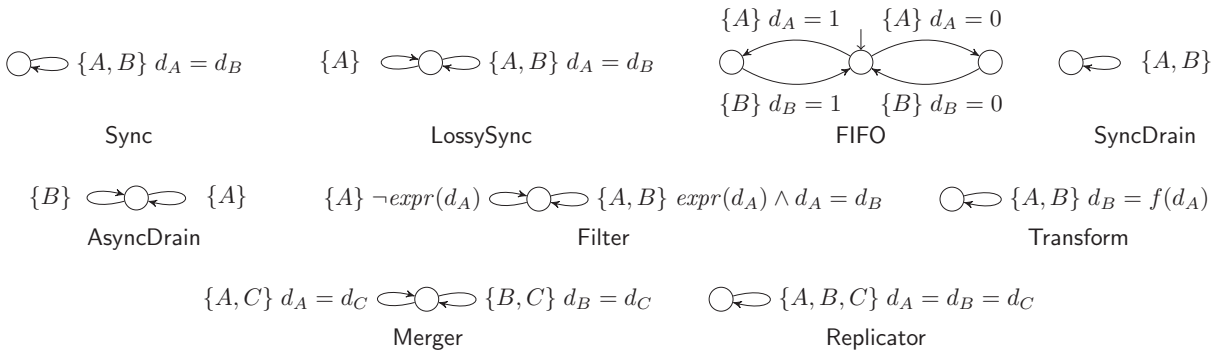


Fig. 2. Constraint automata for basic Reo channels and nodes

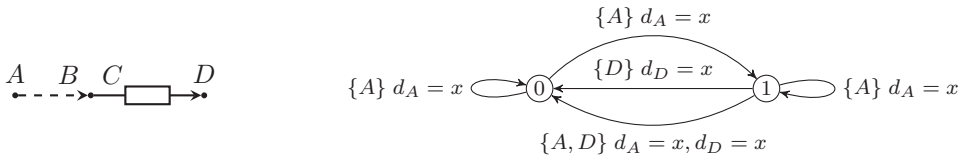


Fig. 3. LossyFIFO connector and its constraint automaton

The definition of a timed constraint automaton is similar to the definition of standard timed automata [AD94]. However, in contrast to the usual timed automata, TCA transition labels have three constituents: (i) synchronization constraints that represent the set of ports where dataflow is observed simultaneously, (ii) data constraints that enable these transitions and, finally, (iii) clock constraints.

A TCA for a *timer with off and reset option* channel is shown in Figure 4. In this model, the state s represents a timer which is switched off while the state $\overline{s}$ corresponds to the timer being switched on.

2.3. Context-Sensitive Semantics for Reo

Constraint automata in their basic form cannot express the behavior of so-called context-dependent channels. An elemental example of such a channel is a LossySync channel that loses a data item only if the environment or subsequent channels are not ready to consume it. Note that the constraint automaton for the LossyFIFO connector in Figure 3, obtained using the product operator of constraint automata for the LossySync and FIFO channels, fails to express the context-sensitivity of the LossySync: even when the LossyFIFO is in the initial state, i.e., the buffer of the FIFO channel is empty and ready to consume a data item, the circuit can loose its input on the source port A. In a correct constraint automata model, dataflow on a port A in the initial state should always lead to a change of the state which signals that the data item was accepted and stored in the buffer. Several models have been proposed to overcome this problem.

Mousavi et al. [MSA06] propose a structural operational semantics for Reo. In this work, context-dependent behavior of the LossySync channel is captured using maximal progress that helps to eliminate undesired behaviors of the circuits. However, this solution cannot express preferences between unrelated

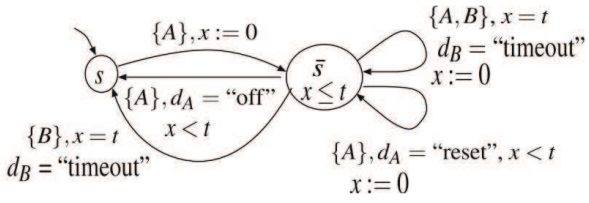
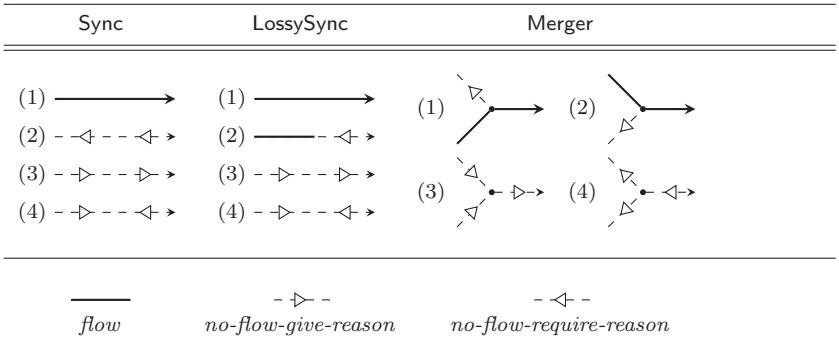


Fig. 4. Timed constraint automaton for the timer channel with off and reset options

Table 1. Examples of coloring semantics for Reo channels and nodes



behaviors and relies on a global rule rather than the compositional construction of an appropriate model. Intentional automata [Cos10] distinguish two sets of ports in their transition labels, a request set and a firing set. The request set models the context, i.e., the readiness of the channel ports to accept/dispense data, while the firing set models the actual flow of data through the circuit ports. Accounting for the requests that have arrived but have not been fired yet introduces additional states in the model. Due to this fact, intentional automata rapidly become large and difficult to manipulate. Reo automata [BCS09] can express context dependency using a request and firing set as in intentional automata. However, in this model the information about pending requests is not stored in states but instead it is calculated on the fly from the transition labels. This makes Reo automata significantly more compact in terms of the number of states and transitions, but the computational complexity of their composition operation is higher.

Connector coloring [CCA07] describes the behavior of Reo in a compositional fashion by coloring the parts of the circuit with and without dataflow using different colors that match on connected ports. When three colors are used, the model captures context-dependent behavior by propagating negative information about the exclusion of dataflow through the connector. This model is used currently as a theoretical basis for Reo circuit animation and simulation tools. The basic idea in this model is to associate flow and no-flow colors to channel ends. Clarke et al. show in [CCA07] that one *flow* color and two *no-flow* colors are sufficient to model context-dependency as, for instance, required by the LossySync. See also [JKA11].

Valid behaviors of channels are then expressed as colorings of their respective ends. Table 1 shows the colorings of the Sync, LossySync and Merger primitives. Note that the colors are always read from the perspective of the adjacent nodes. For instance, in coloring (2) of the Sync, the sink node gives a reason for no flow, whereas the source node requires a reason. This models the behavior where data is available at the source end but the receiver at the sink end is not ready to accept data. Similarly, in coloring (3) there is no flow, because there is no data available at the source end. Finally, coloring (4) models the situation where no data is available and the receiver is also not ready to accept any data.[3]

The LossySync differs from the Sync channel only in one coloring, i.e. coloring (2) where the sink node is not ready to accept data, but there is data available at the source end. In this situation the LossySync permits flow at the source end and loses the data item. Otherwise, no-flow behaviors are possible only when no data is available at the source end.

Reo nodes are encoded in the same way as channels in the coloring semantics. As usual, we build nodes out of mergers and replicators. Table 1 shows the valid colorings of the Merger primitive. An interesting fact here is that intuitively the colorings allow a propagation of no-flow reasons through the connector. Note also that it is sufficient to allow no-flow reasons from both sides in channels only, which leads to a smaller number of coloring in the nodes.

2.4. Action Constraint Automata

Expressing the behavior of Reo connectors is orthogonal to estimating the end-to-end quality of the communication protocol that they implement. A number of semantic models of Reo, most notably, constraint

[3] This behavior is implied by the so-called *flip-rule* in [CCA07].

and intentional automata, have been extended with the information to capture the QoS characteristics of the channels and their composition metrics [ACSM07, ACM+09]. However, these extensions assume that the QoS characteristics do not affect the behavior of a circuit. In our recent work [KCA10], we argued that data transfer delays are important for circuit behavior and accommodating them properly requires an appropriate formal model. Action constraint automata (ACA) is a semantic model for Reo, which, in contrast to constraint automata in their classic form, distinguishes several kinds of actions triggered on channel ports to signal the state changes of the channel. Formally, ACA can be defined as follows:

Definition 2.3 (Action constraint automaton (ACA)). An action constraint automaton $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0)$ consists of a set of states S, a set of action names $\mathcal{N}$ derived from a set of port names $\mathcal{N}$ and a set of admissible action types $\mathcal{T}$, 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 introduce an injective function $act : \mathcal{N} \times \mathcal{T} \rightarrow \mathcal{N}$ to define action names for each pair of a port name and an action type observed on the port. For example, the function $act(m, \alpha) = \alpha \cdot m$, for $m \in \mathcal{N}, \alpha \in \mathcal{T}$, where '$\cdot$' is a standard concatenation operator, can be used to obtain a set of unique action names given sets of distinctive Reo port names and types of observable actions. A product of ACA is defined analogously to the product of ordinary CA, cf. [KCA10].

Since time is required by a channel for its internal coordination and to transfer data, it may happen that the channel is still busy while other requests arrive at the source ports of the circuit. There is no reason why the channels that are not busy at the moment should not process the arrived requests. Therefore, we consider two actions, namely, a 'block' action and its dual an 'unblock' action which are used to establish port communication within a single transaction and release channel ports involved in such a transaction, respectively. For example, to show that a Sync channel with source end A and sink end B is ready to accept and dispense data, 'block' actions bA and bB occur simultaneously. After the data transfer is finished, the channel returns to its initial state when both ports are released and 'unblock' actions uA and uB are observed. To represent dataflow in complex synchronous circuits, in addition to the 'block' and 'unblock' actions, we introduce 'start' and 'finish' actions which are used to represent the start and the end of dataflow through a blocked channel port. Thus, we use the set of action types $\mathcal{T} = \{b, s, f, u\}$, where b stands for the 'block', s for the 'start', f for the 'finish' and u for the 'unblock' action type. The sequence of the aforementioned four actions is observed at each Reo port. Before the start of each transition, ports participating in this transition must be blocked. Then, the data transfer starts. After some time t, which represents the delay in the channel, the 'finish' action occurs to signal that the data transfer is over. Finally, the 'unblock' action releases the channel port, subsequent to which it can be coopted to perform another communication.

Figure 5 shows the semantics of the basic Reo channels with explicit modeling of internal blocking and dataflow within each channel. After blocking actions have occurred in the Sync channel, both its ports start to accept data. This is represented by the simultaneous occurrence of the actions sA and sB. Similarly, after the data transfer is finished, actions fA and fB are observed. For the SyncDrain channel, as usual, we require that its ports are blocked and unblocked simultaneously, while the actual data transfer through the two ports start and end independently, i.e., all interleavings of action pairs (sA, fA) and (sB, fB) are allowed. The semantics of the Merger and the Replicator nodes is defined in a similar way. We assume that, in contrast to channels, the data transfer through a node is instantaneous, i.e., dataflow starts and finishes at the same time. For the scenarios where the time for data replication is significant and cannot be neglected, automata with two different transitions to signal the start and the end of dataflow should be used.

By synchronizing 'finish' actions observed on sink ends with 'start' actions observed on the source ends, we can model sequential flow of data in synchronous regions. Given two action constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ for each pair $X \in \mathcal{N}_1, Y \in \mathcal{N}_2$ of joint ports, where X is a sink port, and Y is a source port, the following pairs of actions happen synchronously:

$$\{\, (act(X, b), act(Y, b)),\ (act(X, u), act(Y, u)),\ (act(X, f), act(Y, s)) \,\}.$$

This approach is compliant with the two principles introduced in [ACM+09], namely, that (i) a data-flow in a channel takes place from its input port to its output port, and (ii) mixed nodes receive and send data instantaneously. Thus, ACA is a suitable compositional model for reasoning about dataflow in Reo circuits with timed delays.

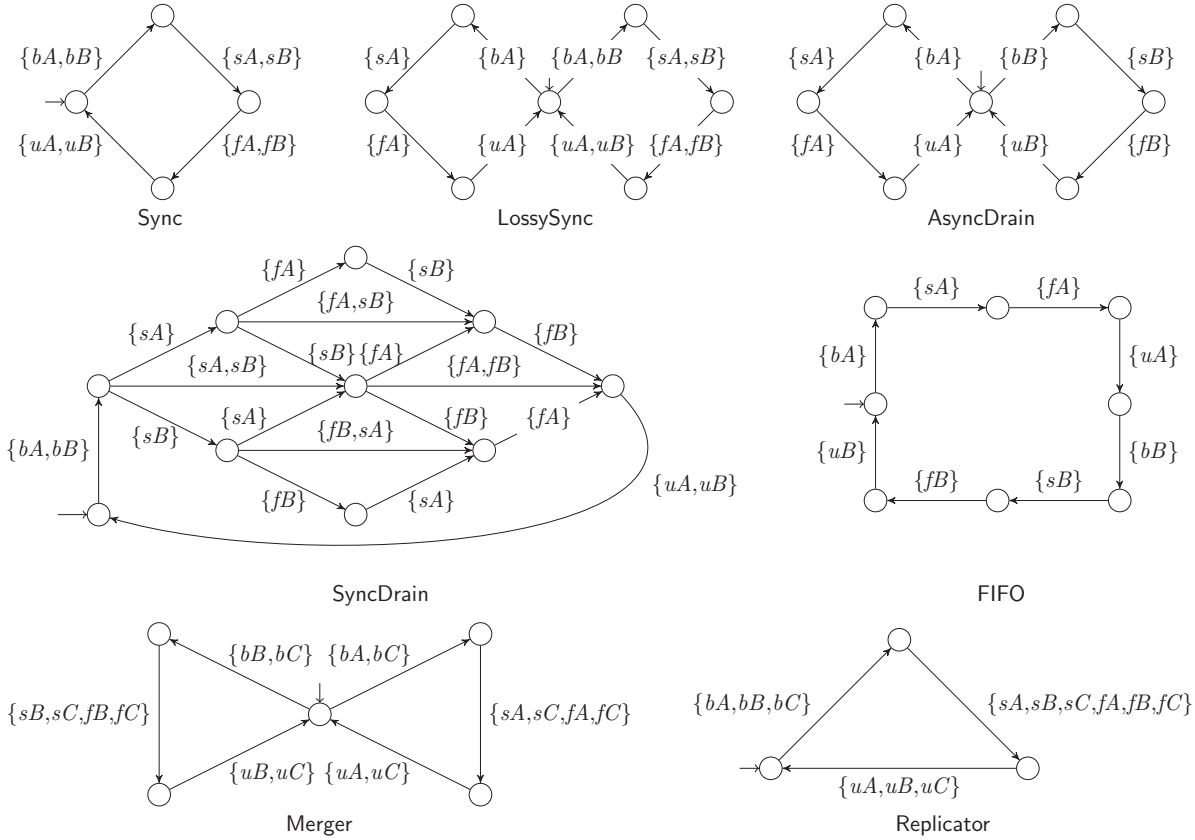


Fig. 5. Action constraint automata for basic Reo channels and nodes

3. Encoding of Reo in mCRL2

One of the strengths of Reo as a workflow modeling language is its being amenable to formal verification. Currently, the most feature-complete and efficient way to analyze a Reo process models is to generate a mCRL2 specification and to use the corresponding verification facilities. In this section, we briefly describe mCRL2 and its application to model checking Reo. For more details on mCRL2 we refer to [Gea07] and to the mCRL2 website.[4]

The basic notion in mCRL2 is the action. Actions represent atomic events and can be parametrized with data. Actions in mCRL2 can be synchronized. In this case, we speak of multiactions which are constructed from other actions or multiactions using the so-called synchronization operator $|$, such as the multiaction $a|b|c$ of simultaneously performing the actions a, b and c. The special action τ (tau) is used to refer to an internal, unobservable action. It cannot be part of a multiaction. Processes are defined by process expressions, which are compositions of actions and multiactions using a number of operators. The basic operators include

- *deadlock* or *inaction* δ, which does not display any behavior;
- *alternative composition*, written as $p + q$, which represents a non-deterministic choice between the processes p and q;
- *sequential composition*, written $p \cdot q$, which means that q is executed after p, assuming that p terminates;
- the *conditional operator* or *if-then-else* construct, written as $c \to p \diamond q$, where c is a data expression that evaluates to true or false;

[4] http://mcrl2.org/mcrl2/wiki/index.php

$$
\begin{array}{rl}
\mathsf{Sync} & = \Sigma_{d:Data}\ A(d)|B(d) \cdot \mathsf{Sync} \\
\mathsf{LossySync} & = \Sigma_{d:Data}\ (A(d)|B(d) + A(d)) \cdot \mathsf{LossySync} \\
\mathsf{SyncDrain} & = \Sigma_{d_1,d_2:Data}\ A(d_1)|B(d_2) \cdot \mathsf{SyncDrain} \\
\mathsf{AsyncDrain} & = \Sigma_{d:Data}\ (A(d) + B(d)) \cdot \mathsf{AsyncDrain} \\
\mathsf{FIFO}(f : DataFIFO) & = \Sigma_{d:Data}(isEmpty(f) \rightarrow A(d) \cdot \mathsf{FIFO}(full(d)) \diamond B(e(f)) \cdot \mathsf{FIFO}(empty)) \\
\mathsf{Filter} & = \Sigma_{d:Data}\ (expr(d) \rightarrow A(d)|B(d) \diamond A(d)) \cdot \mathsf{Filter} \\
\mathsf{Transform} & = \Sigma_{d:Data}\ A(d)|B(f(d)) \cdot \mathsf{Transform} \\
\hline
\mathsf{Merger} & = \Sigma_{d:Data}\ (A(d)|C(d) + B(d)|C(d)) \cdot \mathsf{Merger} \\
\mathsf{Replicator} & = \Sigma_{d:Data}\ A(d)|B(d)|C(d) \cdot \mathsf{Replicator}
\end{array}
$$

Table 2. mCRL2 encoding for channels and nodes: CA semantics

- *summation* $\Sigma_{d:D}\ p$ where p is a process expression in which the data variable d may occur, used to quantify over a data domain D;
- the *at operator* $a@t$, which indicates that the action a happens at time t;
- *parallel composition* or *merge* $p \parallel q$, which interleaves and synchronizes the multiactions of p with those of q, where synchronization is governed by a communication function (see below);
- the *encapsulation operator* $\partial_H(p)$, where H is a set of action names that are not allowed to occur;
- the *renaming operator* $\rho_R(p)$, where R is a set of renamings of the form $a \rightarrow b$, meaning that every occurrence of action a in p is replaced by the action b;
- the *communication operator* $\Gamma_C(p)$, where C is a set of communications of the form $a_0|...|a_n \mapsto c$, which means that every group of actions $a_0|...|a_n$ within a multiaction is replaced by the action c.

The mCRL2 language provides a number of built-in datatypes (e.g., boolean, natural, integer) with the usual arithmetic operations predefined. Moreover, a definition mechanism for abstract datatypes allows users to declare new types, also referred to as sorts. An arbitrary structured type in mCRL2 can be declared by a construct of the form

$$\textbf{sort}\ S = \textbf{struct}\ c_1(p_1^1{:}S_1^1,\ \ldots,\ p_1^{k1}{:}S_1^{k1})?r_1\ \mid\ \ldots\ \mid\ c_n(p_n^1{:}S_n^1,\ \ldots,\ p_n^{kn}{:}S_n^{kn})?r_n\,;$$

It defines the type S together with constructors $c_i\colon S_i^1 \times \ldots \times S_i^{ki} \rightarrow S$, projections $p_i^j\colon S \rightarrow S_i^j$, and type recognition functions $r_i\colon S \rightarrow Bool$.

The mCRL2 toolset allows users to verify software models specified in the mCRL2 language. The toolset includes a tool for converting mCRL2 specifications into linear form (a compact symbolic representation of the corresponding labeled transition system (LTS) to speed up subsequent manipulations), a tool for generating explicit LTSs from linear process specifications (LPS), tools for optimizing and visualizing these LTSs, and many other useful facilities. A detailed overview of the available software can be found at the mCRL2 web site.

For model checking, system properties are specified as formulae in a variant of the modal μ-calculus extended with regular expressions, data and time. In combination with an LPS, such a formula is transformed into a parametrized boolean equation system (PBES) and can be solved with the appropriate tools from the toolset. Analysis of LTSs, in particular, deadlock detection or checking for the presence or absence of certain actions, is also possible.

3.1. Constraint Automata Mapping

A constraint automaton is essentially an LTS with labels representing two kinds of constraints: synchronization and data constraints. Synchronization constraints represent sets of Reo port names where data flow is observed simultaneously during a transition. Data constraints model conditions on transition enactment and show data assignments on port names. Therefore, mCRL2 models for Reo circuits can be generated in the following way: a process for each channel is defined; its observable events, i.e., data flow on the channel ends, are represented as atomic actions, while data items observed at these ports are modeled as parameters of these actions. Analogously, we introduce a process for every node and actions for all channel ends meeting at the node. The encodings for the basic Reo channels and nodes are depicted in Table 2.

Given process definitions for all channels and nodes, a joint process that models a complete Reo connector

is built by forming a parallel composition of these processes and synchronizing actions for coinciding channel/node ends. Channel/node end synchronization is enforced using the mCRL2 operators communication and encapsulation. A process for a Reo circuit is formed by synchronizing actions corresponding to joint channel ends at a node in a parallel composition of processes corresponding to channels and nodes of the circuit.

Let us first consider the case without data.For example, a mCRL2 process for the replicator circuit in Figure 1 can be formed from three synchronous channels

$$\mathsf{Sync1} = A|X_1 \cdot \mathsf{Sync1}\,, \quad \mathsf{Sync2} = Y_1|B \cdot \mathsf{Sync2}\,, \quad \mathsf{Sync3} = Z_1|C \cdot \mathsf{Sync3}$$

and a replicator node

$$\mathsf{ReplicatorNode} = X_2|Y_2|Z_2 \cdot \mathsf{ReplicatorNode}$$

applying the communication and blocking operators to their parallel composition:

$$\mathsf{ReplicatorNode} = \partial_{\{X_1,Y_1,Z_1,X_2,Y_2,Z_2\}}\big(\,\Gamma_{\{X_1|X_2\to\tau,Y_1|Y_2\to\tau,Z_1|Z_2\to\tau\}}(\mathsf{Sync1} \parallel \mathsf{Sync2} \parallel \mathsf{Sync3} \parallel \mathsf{ReplicatorNode}\,)\,\big);$$

Here we assume that the sink end X_1 of the channel Sync1 is connected to the source end X_2 of the node ReplicatorNode, while sink ends Y_2 and Z_2 of the node are connected to source ends Y_1 and Z_1 of channels Sync2 and Sync3.

Although it is possible to describe any circuit as a single process using the principle above, state space explosion may occur since many non-deterministic unfoldings with unmatched internal synchronization actions will be caught at a late stage. To overcome this problem, we seek to minimize the intermediate state spaces during the processing of the mCRL2 specification by building the process for the whole Reo connector in a stepwise fashion – exploiting the graph structure of the circuit. We create a process for a Reo circuit by adding nodes, as well as channels connected to them, one-by-one.

More precisely, consider a Reo network $RC = (V, E)$ as a graph with nodes in V and edges in E. We use $v \in V$, $e \in E$. We assume the predicate $src(v, e)$ to hold if the edge e has a source end in RC that is incident to the node v. Similarly, the predicate $snk(v, e)$ holds if e has a sink end in RC at v. The mCRL2 process $\mathsf{Chan}(e)$ is the encoding of the corresponding channel in RC, see Table 2 (ignoring the data for the moment). If the edge e joins the nodes u and v, we require that the actions $X''_{u,e}$ and $X''_{v,e}$ are precisely the ones of $\mathsf{Chan}(e)$. The mCRL2 process $\mathsf{Node}(v)$ is defined as

$$\mathsf{Node}(v) \;=\; \textstyle\sum_{e:src(v,e)} (X'_{v,e} \mid \otimes_{e:snk(v,e)} X'_{v,e}) \cdot \mathsf{Node}(v)$$

Here, we use $\otimes_{i\in I} X_i$ to denote the multiaction comprised of the actions X_i, $i \in I$. We define the predicate $lnk(e, v, w)$ to be true if the edge e is incident to the node v but not incident to a node in the string $w \in V^*$. Now, for a string w of nodes the process $P(w)$ is inductively given by

$$\begin{aligned} P(\varepsilon) &= \delta \\ P(w \cdot v) &= \partial_{H_v}(\Gamma_{C_v}(\, P(w) \parallel \mathsf{Node}(v) \parallel \textstyle\prod_{e:lnk(e,v,w)} \mathsf{Chan}(e)\,)) \end{aligned}$$

where $H_v = \{X'_{v,e}, X''_{v,e} \mid e \in E,\ lnk(e, v, W)\}$ and $C_v = \{X'_{v,e} \mid X''_{v,e} \to X_{v,e} \mid e \in E,\ lnk(e, v, w)\}$. The notation $\prod_{i\in I} P_i$ denotes the parallel composition of the processes P_i, $i \in I$.
Thus, for the nodes v_1, v_2, v_3 in the excerpt of a Reo connector in Figure 6 we have the following:

$$\begin{aligned} \mathsf{Node}(v_1) &= (X'_{v_1,e_1}|X'_{v_1,e_3}|X'_{v_1,e_4}|X'_{v_1,e_5}) \cdot \mathsf{Node}(v_1) + (X'_{v_2,e_1}|X'_{v_1,e_3}|X'_{v_1,e_4}|X'_{v_1,e_5}) \cdot \mathsf{Node}(v_1) \\ \mathsf{Node}(v_2) &= (X'_{v_2,e_3}|X'_{v_2,e_6}|X'_{v_2,e_7}|X'_{v_2,e_8}) \cdot \mathsf{Node}(v_2) \\ \mathsf{Node}(v_3) &= (X'_{v_3,e_4}|X'_{v_3,e_6}|X'_{v_3,e_9}) \cdot \mathsf{Node}(v_3) \end{aligned}$$

The two non-deterministic options in the defining clause of $\mathsf{Node}(v_1)$ at the RHS stem from the two incoming edges, e_1 and e_2 for the node v_1. Following the encoding for the primitive channels in Table 2 we obtain for the channels e_3, e_4, e_5 the mCRL2-processes

$$\begin{aligned} \mathsf{Chan}(e_3) &= (\,(X''_{v_1,e_3}|X''_{v_2,e_3}) + X''_{v_1,e_3}\,) \cdot \mathsf{Chan}(e_3) \\ \mathsf{Chan}(e_4) &= (X''_{v_1,e_4}|X''_{v_3,e_4}) \cdot \mathsf{Chan}(e_4) \\ \mathsf{Chan}(e_6) &= (X''_{v_2,e_6}|X''_{v_3,e_6}) \cdot \mathsf{Chan}(e_5) \end{aligned}$$

while the encodings of the other channels e_1, e_2, e_5, e_7, e_8 and e_9 will involve, among others, X''_{v_1,e_1}, X''_{v_1,e_2},

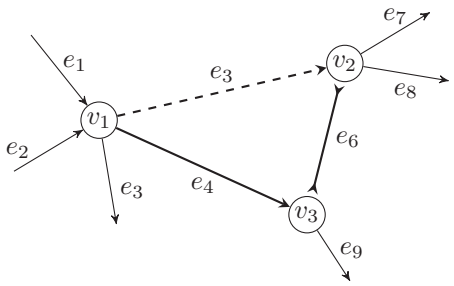


Fig. 6. A fragment of a Reo connector

X''_{v_1,e_5}, X''_{v_2,e_7}, X''_{v_2,e_8} and X''_{v_3,e_9}. Following the construction above, expanding v_1 first, we get

$$\begin{array}{rcl} P(v_1) & = & \partial_{H_{v_1}}(\,\Gamma_{C_{v_1}}(\mathsf{Node}(v_1) \parallel \mathsf{Chan}(e_1) \parallel \mathsf{Chan}(e_2) \parallel \mathsf{Chan}(e_3) \parallel \mathsf{Chan}(e_4) \parallel \mathsf{Chan}(e_5))\,) \text{ with} \\ H_{v_1} & = & \{\, X'_{v_1,e}, X''_{v_1,e} \mid e = e_1, e_2, e_3, e_4, e_5 \,\} \\ C_{v_1} & = & \{\, X'_{v_1,e} | X''_{v_1,e} \mapsto X_{v_1,e} \mid e = e_1, e_2, e_3, e_4, e_5 \,\} \end{array}$$

Here we have used ACP's equational law $\delta \parallel Q = Q$ stating that it is safe to leave out δ, modulo strong bisimulation. Adding v_2 yields

$$\begin{array}{rcl} P(v_1 \cdot v_2) & = & \partial_{H_{v_2}}(\,\Gamma_{C_{v_2}}(P(v_1) \parallel \mathsf{Node}(v_2) \parallel \mathsf{Chan}(e_6) \parallel \mathsf{Chan}(e_7) \parallel \mathsf{Chan}(e_8))\,) \text{ with} \\ H_{v_2} & = & \{\, X'_{v_2,e}, X''_{v_2,e} \mid e = e_6, e_7, e_8 \,\} \\ C_{v_2} & = & \{\, X'_{v_2,e} | X''_{v_2,e} \mapsto X_{v_2,e} \mid e = e_6, e_7, e_8 \,\} \end{array}$$

Channel e_3 need not to be added again; it is already incorporated in the process $P(v_1)$. Consistent with this, $lnk(e_3, v_2, v_1)$ does not hold; although e_3 is incident upon v_2, it is incident upon v_1 too, hence $lnk(e_3, v_2, v_1)$ is false. Similarly, as a final illustration of the construction, we have

$$\begin{array}{rcl} P(v_1 \cdot v_2 \cdot v_3) & = & \partial_{H_{v_3}}(\,\Gamma_{C_{v_3}}(P(v_1 \cdot v_2) \parallel \mathsf{Node}(v_3) \parallel \mathsf{Chan}(e_9)\,) \text{ with} \\ H_{v_3} & = & \{\, X'_{v_3,e_9}, X''_{v_3,e_9} \,\} \\ C_{v_3} & = & \{\, X'_{v_3,e_9} | X''_{v_3,e_9} \mapsto X_{v_3,e_9} \,\} \end{array}$$

as channel e_9 is the only channel not covered by $P(v_1 \cdot v_2)$ yet.

Directly using the operational semantics of `mCRL2` [Gea07], we can show that the following equivalences with respect to strong bisimulation $\leftrightarrows$ [Mil89] are valid.

Proposition 3.1. Let $H_i = \{X'_k, X''_k \mid k \in K_i\}$, $C_i = \{X'_k \mid X''_k \to X_k \mid k \in K_i\}$, for $i = 1, 2$.

(a) If $H_1 \cap H_2 = \emptyset$ then $\partial_{H_1}(\Gamma_{C_1}(\partial_{H_2}(\Gamma_{C_2}(P)))) \leftrightarrows \partial_{H_2}(\Gamma_{C_2}(\partial_{H_1}(\Gamma_{C_1}(P))))$ for any `mCRL2` process P.

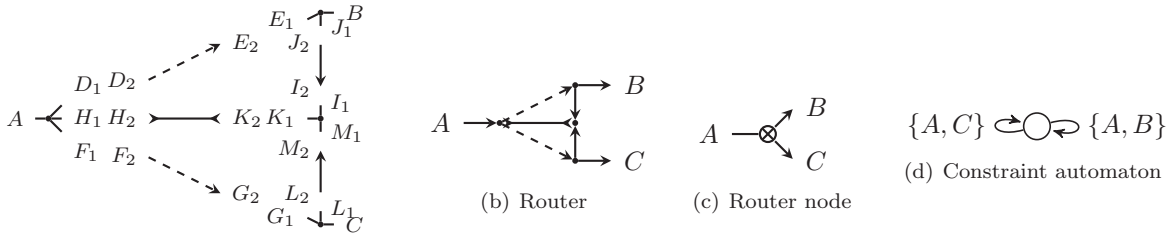
(b) If $H_1 \cap act(P) = \emptyset$ then $\partial_{H_1}(\Gamma_{C_1}(P)) \leftrightarrows P$ for any `mCRL2` process P, with $act(P)$ the action alphabet of P.

(c) $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$ for all `mCRL2` processes P, Q, R. □

Using the properties above we can derive that the construction of a `mCRL2` process for a Reo connector is independent, modulo strong bisimulation, of the order of nodes chosen.

Theorem 3.1. Let RC be a Reo connector with nodes $v_1, \ldots v_n$ and $\pi\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ a permutation, then $P(v_1 v_2 \cdots v_n) \leftrightarrows P(v_{\pi(1)} v_{\pi(2)} \cdots v_{\pi(n)})$.

Proof. As any permutation can be represented as a composition of transpositions, it suffices to show, by the various properties of the parallel, communication and encapsulation operator, that $P(w \cdot u \cdot v) = P(w \cdot v \cdot u)$ for



(a) Decomposed router (b) Router (c) Router node (d) Constraint automaton

Fig. 7. A router connector and associated constraint automaton

any string of nodes w over RC not containing the different nodes u and v. We have

$$
\begin{aligned}
&P(w{\cdot}u{\cdot}v)\\
&= \quad \partial_{H_v}(\Gamma_{C_v}(\,\partial_{H_u}(\Gamma_{C_u}(\,P(w) \parallel \mathsf{Node}(u) \parallel \textstyle\prod_{e:lnk(e,u,w)} \mathsf{Chan}(e)\,))\\
&\qquad \parallel \mathsf{Node}(v) \parallel \textstyle\prod_{e:lnk(e,v,w\cdot u)} \mathsf{Chan}(e)\,))\\
&= \quad \partial_{H_v}(\Gamma_{C_v}(\,\partial_{H_u}(\Gamma_{C_u}(\,P(w) \parallel \mathsf{Node}(u) \parallel \textstyle\prod_{e:lnk(e,u,w)} \mathsf{Chan}(e)\\
&\qquad \parallel \mathsf{Node}(v) \parallel \textstyle\prod_{e:lnk(e,v,w\cdot u)} \mathsf{Chan}(e)\,))\,)) \qquad \text{by Proposition 3.1b}\\
&= \quad \partial_{H_u}(\Gamma_{C_u}(\,\partial_{H_v}(\Gamma_{C_v}(\,P(w) \parallel \mathsf{Node}(u) \parallel \textstyle\prod_{e:lnk(e,u,w)} \mathsf{Chan}(e)\\
&\qquad \parallel \mathsf{Node}(v) \parallel \textstyle\prod_{e:lnk(e,v,w\cdot u)} \mathsf{Chan}(e)\,))\,)) \qquad \text{by Proposition 3.1a}\\
&= \quad \partial_{H_u}(\Gamma_{C_u}(\,\partial_{H_v}(\Gamma_{C_v}(\,P(w) \parallel \mathsf{Node}(u) \parallel \textstyle\prod_{e:lnk(e,u,w\cdot v)} \mathsf{Chan}(e) \parallel \prod_{e:e \text{ joins } u \text{ and } v} \mathsf{Chan}(e)\\
&\qquad \parallel \mathsf{Node}(v) \parallel \textstyle\prod_{e:lnk(e,v,w\cdot u)} \mathsf{Chan}(e)\,))\,)) \qquad lnk(e,u,w) \Leftrightarrow lnk(e,u,w{\cdot}v) \vee e \text{ joins } u \text{ and } v\\
&= \quad \partial_{H_u}(\Gamma_{C_u}(\,\partial_{H_v}(\Gamma_{C_v}(\,P(w) \parallel \mathsf{Node}(v) \parallel \textstyle\prod_{e:lnk(e,v,w\cdot u)} \mathsf{Chan}(e) \parallel \prod_{e:e \text{ joins } u \text{ and } v} \mathsf{Chan}(e)\\
&\qquad \parallel \mathsf{Node}(u) \parallel \textstyle\prod_{e:lnk(e,u,w\cdot v)} \mathsf{Chan}(e)\,))\,)) \qquad \text{by Proposition 3.1c}\\
&= \quad \partial_{H_u}(\Gamma_{C_u}(\,\partial_{H_v}(\Gamma_{C_v}(\,P(w) \parallel \mathsf{Node}(v) \parallel \textstyle\prod_{e:lnk(e,v,w\cdot u)} \mathsf{Chan}(e) \parallel \prod_{e:e \text{ joins } u \text{ and } v} \mathsf{Chan}(e)\,))\\
&\qquad \parallel \mathsf{Node}(u) \parallel \textstyle\prod_{e:lnk(e,u,w\cdot v)} \mathsf{Chan}(e)\,)) \qquad \text{by Proposition 3.1a}\\
&= \quad \partial_{H_u}(\Gamma_{C_u}(\,\partial_{H_v}(\Gamma_{C_v}(\,P(w) \parallel \mathsf{Node}(v) \parallel \textstyle\prod_{e:lnk(e,v,w)} \mathsf{Chan}(e)\,))\\
&\qquad \parallel \mathsf{Node}(u) \parallel \textstyle\prod_{e:lnk(e,u,w\cdot v)} \mathsf{Chan}(e)\,)) \qquad \text{definition } lnk(e,v,w)\\
&= \quad P(w{\cdot}v{\cdot}u)
\end{aligned}
$$

as was to be checked. □

3.2. Example Encodings

As a concrete example without data, consider the router circuit shown in Figure 7(b). We first generate a process

$$P_0 = \partial_{\{D_1,D_2,F_1,F_2,H_1,H_2\}}\big(\,\Gamma_{\{D_1|D_2\to D,F_1|F_2\to F,H_1|H_2\to H\}}(\mathsf{Node1} \parallel \mathsf{LossySync1} \parallel \mathsf{LossySync2} \parallel \mathsf{SyncDrain1}\,)\big)$$

which connects the input node

$$\mathsf{Node1} = A|D_1|F_1|H_1 \cdot \mathsf{Node1}$$

to the source ends of the channels

$$\mathsf{LossySync1} = (D_2 + D_2|E_2) \cdot \mathsf{LossySync1}$$
$$\mathsf{LossySync2} = (F_2 + F_2|G_2) \cdot \mathsf{LossySync2}$$
$$\mathsf{SyncDrain1} = H_2|K_2 \cdot \mathsf{SyncDrain1}$$

Then, we define the process

$$P_1 = \partial_{\{E_1,E_2,J_1,J_2\}}\big(\,\Gamma_{\{E_1|E_2\to E,J_1|J_2\to J\}}(P_0 \parallel \mathsf{Node2} \parallel \mathsf{Sync1}\,)\big)$$

which connects the next node

$\mathsf{Node2} = E_1|J_1|B \cdot \mathsf{Node2}$

and the associated channel

$\mathsf{Sync1} = J_2|I_2 \cdot \mathsf{Sync2}$

to the process P_0. Adding the middle merge node

$\mathsf{Node3} = (K_1|M_1 + K_1|I_1) \cdot \mathsf{Node3}$

and the remaining synchronous channel

$\mathsf{Sync2} = M_2|L_2 \cdot \mathsf{Sync3}$

to the process P_1 yields

$P_2 = \partial_{\{M_1,M_2,K_1,K_2,I_1,I_2\}}\big(\Gamma_{\{M_1|M_2\to E,K_1|K_2\to K,I_1|I_2\to I\}}(P_1 \parallel \mathsf{Node3} \parallel \mathsf{Sync2})\big)$

Finally, adding the node

$\mathsf{Node4} = G_1|L_1|C \cdot \mathsf{Node4}$

constructing the process

$P_3 = \partial_{\{G_1,G_2,L_1,L_2\}}\big(\Gamma_{\{G_1|G_2\to G,L_1|L_2\to L\}}(P_2 \parallel \mathsf{Node4})\big)$

completes the mCRL2 specification, which in total reads as

$$P_0 = \partial_{\{D_1,D_2,F_1,F_2,H_1,H_2\}}\big(\Gamma_{\{D_1|D_2\to D,F_1|F_2\to F,H_1|H_2\to H\}}(\mathsf{Node1} \parallel \mathsf{LossySync1} \parallel \mathsf{LossySync2} \parallel \mathsf{SyncDrain1})\big)$$
$$P_1 = \partial_{\{E_1,E_2,J_1,J_2\}}\big(\Gamma_{\{E_1|E_2\to E,J_1|J_2\to J\}}(P_0 \parallel \mathsf{Node2} \parallel \mathsf{Sync1})\big)$$
$$P_2 = \partial_{\{M_1,M_2,K_1,K_2,I_1,I_2\}}\big(\Gamma_{\{M_1|M_2\to E,K_1|K_2\to K,I_1|I_2\to I\}}(P_1 \parallel \mathsf{Node3} \parallel \mathsf{Sync2})\big)$$
$$P_3 = \partial_{\{G_1,G_2,L_1,L_2\}}\big(\Gamma_{\{G_1|G_2\to G,L_1|L_2\to L\}}(P_2 \parallel \mathsf{Node4})\big)$$

In a similar way, we can obtain an encoding taking nodes in a different order, e.g.

$$Q_0 = \partial_{\{G_1,G_2,L_1,L_2\}}\big(\Gamma_{\{G_1|G_2\to G,L_1|L_2\to L\}}(\mathsf{Node4} \parallel \mathsf{LossySync2} \parallel \mathsf{Sync2})\big)$$
$$Q_1 = \partial_{\{I_1,I_2,M_1,M_2,K_1,K_2\}}\big(\Gamma_{\{I_1|I_2\to E,M_1|M_2\to M,K_1|K_2\to K\}}(Q_0 \parallel \mathsf{Node3} \parallel \mathsf{Sync1} \parallel \mathsf{SyncDrain1})\big)$$
$$Q_2 = \partial_{\{E_1,E_2,G_1,G_2\}}\big(\Gamma_{\{E_1|E_2\to E,J_1|J_2\to J\}}(Q_1 \parallel \mathsf{Node2} \parallel \mathsf{LossySync1})\big)$$
$$Q_3 = \partial_{\{D_1,D_2,H_1,H_2,F_1,F_2\}}\big(\Gamma_{\{D_1|D_2\to D,H_1|H_2\to H,F_1|F_2\to F\}}(Q_2 \parallel \mathsf{Node1})\big)$$

As Theorem 3.1 proves, it holds that $P_3 \leftrightarrows Q_3$.

Optionally, the mCRL2 hiding operator can be employed for abstracting from the flow in the internal nodes. We will not pursue this in this paper.

For the treatment of data we assume, in the context of a given connector, a global datatype given as the custom sort *Data* in mCRL2. Given such a datatype, we can use the mCRL2 summation operator to define data dependencies imposed by channels. For the FIFO channel we additionally define the datatype

sort $\mathit{DataFIFO} = \mathbf{struct}\ \mathit{empty}?\mathit{isEmpty} \mid \mathit{full}(e{:}\mathit{Data})?\mathit{isFull}$

which allows us to specify whether the buffer of the channel is empty or full, and if it is full, what value is stored in it.

For dataflow modeling, we introduce a special kind of node, Join, denoted as $\oplus$. This node synchronizes all ends of incoming channels, forms a tuple of data items received and replicates it to the source ends of all outgoing channels. To handle the data structures formed by the Join node, we need to close our global datatype under tupling. Thus, if a circuit needs to coordinate services operating on domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ and contains one or more Join nodes with two incoming ends, we define

sort $\mathit{Data} = \mathbf{struct}\ D_1(e_1{:}\mathcal{D}_1) \mid \ldots \mid D_n(e_n{:}\mathcal{D}_n) \mid \mathit{tuple}(p_1{:}\mathit{Data}, p_2{:}\mathit{Data})$

For example, for a Join node with two incoming channels with sink ports A and B and one outgoing channel with source end C, the mCRL2 encoding is as follows:

$\mathsf{Join} = \Sigma_{d_1,d_2:\mathit{Data}}\, A(d_1)|B(d_2)|C(\mathit{tuple}(d_1,d_2)) \cdot \mathsf{Join};$

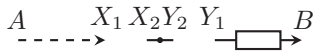


Fig. 8. Decomposed LossyFIFO connector

Note, different from a Merger, the Join requires flow at both it sources ends and dispenses on its sink end the tuple of both incoming data items, rather than one of them.

More generally, when Join nodes with k incoming ends are present in a circuit, $tuple_k(p_1{:}Data, \ldots, p_k{:}Data)$ is added to the definition of the global datatype.

As an example of an encoding of a Reo circuit involving data, we return to the LossyFIFO connector shown in Figure 8. This connector is composed of a LossySync and a FIFO channel connected to a common replication node. The three constituents of the LossyFIFO circuit are translated to the following mCRL2 processes:

$$\begin{array}{rcl} \mathsf{LossySync} & = & \Sigma_{d:Data}\ (A(d)|X_1(d) + A(d)) \cdot \mathsf{LossySync}\,; \\ \mathsf{Node} & = & \Sigma_{d:Data}\ X_2(d)|Y_2(d) \cdot \mathsf{Node}\,; \\ \mathsf{FIFO}(f : DataFIFO) & = & \Sigma_{d:Data}\ isEmpty(f) \rightarrow Y_1(d) \cdot \mathsf{FIFO}(full(d)) \diamond B(e(f)) \cdot \mathsf{FIFO}(empty)\,; \end{array}$$

For obtaining the mCRL2 process for the LossyFIFO connector, we either first form the parallel composition of the three processes above, and force actions corresponding to the connected channel and node ends to communicate:

$$\mathsf{Connector} = \partial_{\{X_1,X_2,Y_1,Y_2\}}(\ \Gamma_{\{X_1|X_2\rightarrow X, Y_1|Y_2\rightarrow Y\}}(\ \mathsf{Node} \parallel \mathsf{LossySync} \parallel \mathsf{FIFO}\ ));$$

Alternatively, following the stepwise construction described above, we exploit the information about the circuit structure, and build up the process for the connector in a stepwise fashion, i.e.,

$$\begin{array}{rcl} \mathsf{Connector1} & = & \partial_{\{Y_1,Y_2\}}(\ \Gamma_{\{Y_1|Y_2\rightarrow Y\}}(\mathsf{Node} \parallel \mathsf{FIFO}))\,; \\ \mathsf{Connector} & = & \partial_{\{X_1,X_2\}}(\ \Gamma_{\{X_1|X_2\rightarrow X\}}(\mathsf{Connector1} \parallel \mathsf{LossySync}))\,; \end{array}$$

This helps in keeping the intermediate state spaces relatively small. Exploiting the topology of the connector to determine an order of the processes significantly decreases the run-time of the mCRL22 linearization algorithm (because of the limited need for alphabet reduction performed internally). A comparison of the actual run-times of the two approaches is discussed in Section 6.

3.3. Timed Constraint Automata Mapping

For the mapping of timer channels to mCRL2, we need to distinguish several data input and output values to represent *off*, *reset* and *timeout* signals that trigger the state change of the timer. This can be done by declaring the following data structure:

$$\textbf{sort}\ DataTimer = \textbf{struct}\ reset?isReset \mid off?isOff \mid timeout?isTimeOut \mid other(e{:}Data)?isOther$$

Timer channels behave differently when switched on or switched off. Taking this into account, the *timer with off- and reset options* can be specified as a parametrized process

$$\begin{array}{l} Timer(isOFF{:}Bool, x{:}Real, t{:}Real) = \\ \quad isOFF \rightarrow (\Sigma_{d:DataTimer}\ isOther(d) \rightarrow A(d) \cdot Timer(false, 0, t)) \diamond \\ \quad\quad ((x < t) \rightarrow (\Sigma_{d:DataTimer}\ isReset(d) \rightarrow A(d) \cdot Timer(false, 0, t) + \\ \quad\quad\quad isOff(d) \rightarrow A(d) \cdot Timer(true, x, t)\ + tick@x \cdot Timer(false, x+1, t)) \diamond \\ \quad\quad\quad B(timeout) \cdot Timer(true, x, t)) \end{array}$$

where $isOFF{:}Bool$ indicates whether the timer is off or on, x is the current time, t is the timer delay, A and B are source and sink ends of the channel and the action *tick* occurring at time x represents the progress of time.

3.4. Coloring Semantics Mapping

The coloring semantics of Reo networks deals with context-dependency. Presence and absence of flow at a channel end is represented by one of the three colors: a color for flow, a color for no-flow stemming from the

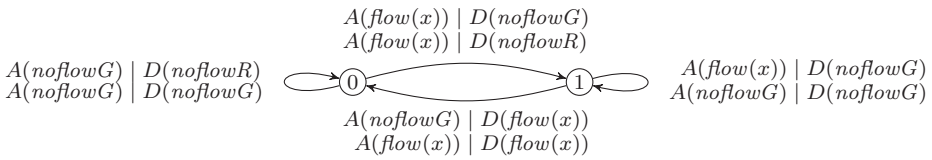


Fig. 9. Coloring encoding of LossyFIFO connector

channel itself, a color for no-flow due to the absence of incoming flow. To incorporate the colorings in our encoding in mCRL2 we simply treat the different colors as data parameters of actions. We therefore introduce a new datatype

$$\textbf{sort}\ \mathit{Colored}\ =\ \textbf{struct}\ \mathit{flow}(\mathit{data}:\mathit{Data})\mid \mathit{noflowG}\mid \mathit{noflowR}$$

where *Data* is the global datatype as introduced in the constraint automata encoding presented in the previous section. The idea is that we explicitly model no-flow actions and wrap actual data items into flow actions. We use here *noflowG* and *noflowR* as abbreviations for respectively *no-flow-give-reason* and *no-flow-require-reason*. With this setup, the encoding of the primitives is straightforward. For instance, the Sync channel is defined as

$$\begin{aligned}\mathsf{Sync} = \big(\Sigma_{d:Data}\ A(\mathit{flow}(d))\mid B(\mathit{flow}(d))\ +\ A(\mathit{noflowR})\mid B(\mathit{noflowG})\ +\\ A(\mathit{noflowG})\mid B(\mathit{noflowR})\ +\ A(\mathit{noflowG})\mid B(\mathit{noflowG})\big)\cdot \mathsf{Sync}\,;\end{aligned}$$

where the various summands correspond to a coloring in Table 1. In the same way, again referring to Table 1, the LossySync can be specified as

$$\begin{aligned}\mathsf{LossySync} = \big(\Sigma_{d:Data}\ A(\mathit{flow}(d))\mid B(\mathit{flow}(d))\ +\ A(\mathit{flow})\mid B(\mathit{noflowG})\ +\\ A(\mathit{noflowG})\mid B(\mathit{noflowR})\ +\ A(\mathit{noflowG})\mid B(\mathit{noflowG})\big)\cdot \mathsf{LossySync}\,;\end{aligned}$$

and, the Merger can be encoded as

$$\begin{aligned}\mathsf{Merger} = \big(\Sigma_{d:Data}\ A(\mathit{flow}(d))\ \mid B(\mathit{noflowG})\mid C(\mathit{flow}(d))\ +\\ \Sigma_{d:Data}\ A(\mathit{noflowG})\mid B(\mathit{flow}(d))\mid C(\mathit{flow}(d))\ +\\ A(\mathit{noflowR})\mid B(\mathit{noflowR})\mid C(\mathit{noflowG})\ +\\ A(\mathit{noflowG})\mid B(\mathit{noflowG})\mid C(\mathit{noflowR})\big)\cdot \mathsf{Merger}\,;\end{aligned}$$

The other channels are encoded analogously.

The LossyFIFO connector shown in Figure 8 is an example where context-dependency is required. Figure 9 displays the generated LTS for the LossyFIFO connector. For simplicity, we show a single transition with several labels instead of several transitions with distinct labels between the same states. The crucial point here is that in the initial state 0, the constraint automata version can lose data (the self loop for state 0 in Figure 3), which is not intended. However, in the coloring encoding, there is no such behavior. Compared to the modeling as a constraint automaton, here we have refined the information labeling the transitions. Note, the refinement is both at the level of the ports involved as in the coloring data attached to the ports.

Using the coloring model we can properly represent context-dependency in mCRL2. In contrast to [CCA07], our encoding also reflects the state of the connectors and can further include data-dependency at the same time. Note also that even though the coloring encoding includes extra transitions for no-flow actions, the number of states is equal to the constraint automata version.

3.5. Action Constraint Automata Mapping

The availability of the synchronization operator and multiactions in mCRL2 makes the translation of ACA to the process algebra mCRL2 straightforward: we simply synchronize simultaneously observed actions in ACA. Table 3 shows the mCRL2 encodings for the basic Reo channels and nodes according to the ACA semantic model. Since data support in the new translation is analogous to the case of the CA-based translation, we omit its discussion here and for simplicity show only the data-agnostic mapping. Note that the expression

Table 3. mCRL2 encoding for channels and nodes: ACA semantics

$$
\begin{aligned}
\mathsf{Sync} &= bA|bB \cdot sA|sB \cdot fA|fB \cdot uA|uB \cdot \mathsf{Sync}\\
\mathsf{LossySync} &= (bA|bB \cdot sA|sB \cdot fA|fB \cdot uA|uB + bA \cdot sA \cdot fA \cdot uA) \cdot \mathsf{LossySync}\\
\mathsf{SyncDrain} &= bA|bB \cdot (sA \cdot (sB \cdot (fA \cdot fB + fB \cdot fA + fA|fB) + fA \cdot sB \cdot fB + sB|fA \cdot fB) +\\
&\qquad sB \cdot (sA \cdot (fA \cdot fB + fB \cdot fA + fA|fB) + fB \cdot sA \cdot fA + sA|fB \cdot fA) +\\
&\qquad sA|sB \cdot (fA \cdot fB + fB \cdot fA + fA|fB)) \cdot uA|uB \cdot \mathsf{SyncDrain}\\
\mathsf{AsyncDrain} &= (bA \cdot sA \cdot fA \cdot uA + bB \cdot sB \cdot fB \cdot uB) \cdot \mathsf{AsyncDrain}\\
\mathsf{FIFO} &= isEmpty(f) \rightarrow bA \cdot sA \cdot fA \cdot uA \cdot \mathsf{FIFO}(full) \diamond bB \cdot sB \cdot fB \cdot uB \cdot \mathsf{FIFO}(empty)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Merger} &= (bA|bC \cdot sA|sC|fA|fC.uA|uC + bB|bC \cdot sB|sC|fB|fC \cdot uB|uC) \cdot \mathsf{Merger}\\
\mathsf{Replicator} &= bA|bB|bC \cdot sA|sB|sC \cdot fA|fB|fC \cdot uA|uB|uC \cdot \mathsf{Replicator}
\end{aligned}
$$

for the SyncDrain channel in the table is equivalent to

$$\mathsf{SyncDrain} = bA|bB \cdot ((sA \cdot fA) \parallel (sB \cdot fB)) \cdot uA|uB \cdot \mathsf{SyncDrain};$$

However, the use of the parallel operator in mCRL2 is restricted because of the difficulties to linearize processes where such an operator occurs in the scope of the sequential, alternative, summation or synchronization operators.

As in the CA approach, we construct nodes compositionally out of the Merger and the Replicator primitives. Given process definitions for all channels and nodes, a joint process that models the complete Reo connector is built by forming a parallel composition of these processes and synchronizing the actions for the coinciding channel/node ends.

4. Correctness of the translation

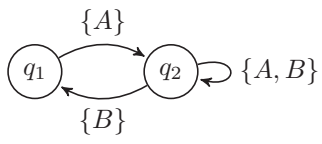
In this section, we address the correctness of the mapping of Reo to mCRL2. For simplicity, here we consider only data-agnostic connectors modeled as constraint automata. In a constraint automaton a single state represents a specific combination of states for the stateful channels of the connector, most notably regarding the FIFO channels in the connector. Extensions with data, time, coloring and actions are treated mutatis mutandis. Also, for the sake of presentation, we consider constraint automata with disjoint sets of port names. Note that this does not affect the generality of our approach as we can always make these sets disjoint by applying an appropriate renaming of ports.

For each state s of a given constraint automaton $\mathcal{A}$, the mCRL2 process $proc(\mathcal{A}, s)$ over the action set $\mathcal{P}(\mathcal{N})$ is given by the transitions that are possible from s, i.e.

$$proc(\mathcal{A}, s) = \textstyle\sum_{s \xrightarrow{N} t} N \cdot proc(\mathcal{A}, t), \tag{1}$$

where $N = \otimes_{x \in N}\, x$ represents the multiaction composed from all ports in the set. Thus, for example, $\otimes_{x \in \{A,B,C\}}\, x = A|B|C$. In this view, it comes natural to have for the synchronization $N_1|N_2$ of actions N_1 and N_2 the union of the underlying port names $N_1 \cup N_2$. If the action N_1 claims flow at the ports of the set N_1 and the action N_2 does so for the ports of the set N_2, supposedly there is flow at the ports of the set $N_1 \cup N_2$.

As an example, consider the synchronous FIFO $\mathcal{F}$ which behaves similarly to the usual FIFO except that it also can dispense a data item from its buffer and simultaneously accept a new one. Its semantics is given by the constraint automaton and the corresponding mCRL2 processes below.



$$
\begin{aligned}
&proc(\mathcal{F}, q_1) = A \cdot proc(\mathcal{F}, q_2)\,;\\
&proc(\mathcal{F}, q_2) = B \cdot proc(\mathcal{F}, q_1) + A|B \cdot proc(\mathcal{F}, q_2)\,;
\end{aligned}
$$

In essence, the translation as discussed in Section 3 recursively decomposes a Reo connector into two subconnectors and puts the mCRL2 processes obtained for these subconnectors in parallel, yielding the process

for the main connector. To prove the correctness of this approach formally, we introduce two operations, a synchronous product $\bowtie_\gamma$ for constraint automata and a synchronized merge $\|_\gamma$ for mCRL2 processes. Thus, given a constraint automaton $\mathcal{A}$ for which we have $\mathcal{A} = \mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2$, we translate the constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$, say into the mCRL2 processes P_1 and P_2, and obtain $P_1 \|_\gamma P_2$ as the translation of $\mathcal{A}$.

Definition 4.1. Let $\mathcal{A}_1 = (S_1, \mathcal{N}_1, \rightarrow_1, s_0^1)$, $\mathcal{A}_2 = (S_2, \mathcal{N}_2, \rightarrow_2, s_0^2)$ be two constraint automata with disjoint sets of port names $\mathcal{N}_1$ and $\mathcal{N}_2$, respectively. A *port synchronization function* $\gamma \colon \mathcal{N} \to \mathcal{N}_1 \times \mathcal{N}_2$ is defined as $\gamma(n) = (\gamma_1(n), \gamma_2(n))$ through the pair of injective functions $\gamma_1 \colon \mathcal{N} \to \mathcal{N}_1$ and $\gamma_2 \colon \mathcal{N} \to \mathcal{N}_2$ that map port names from a new set $\mathcal{N}$ into port names from the sets $\mathcal{N}_1$ and $\mathcal{N}_2$.

Intuitively, $\gamma(z) = (x, y)$ represents a renaming of $x \in \mathcal{N}_1$ and $y \in \mathcal{N}_2$ to the same common element $z \in \mathcal{N}$. This formalizes our use in Section 3 of actions $X'_{v,e}$ and $X''_{v,e}$ of a node v and of an incident channel e, respectively, that synchronize to $X'_{v,e} \mid X''_{v,e} = X_{v,e}$. Thus, for the product of the node and the channel, $\gamma(X_{v,e}) = (\, X'_{v,e},\, X''_{v,e}\,)$. In the context of the port synchronization function γ, we write $\mathcal{N}'_1$ for $\mathcal{N}_1 \backslash \gamma_1[\mathcal{N}]$ and $\mathcal{N}'_2$ for $\mathcal{N}_2 \backslash \gamma_2[\mathcal{N}]$. If, for subsets $N_1 \subseteq \mathcal{N}_1$, $N_2 \subseteq \mathcal{N}_2$, it holds that $\gamma_1^{-1}[N_1] = \gamma_2^{-1}[N_2]$ we write

$$N_1 \mid_\gamma N_2 = (N_1 \cap \mathcal{N}'_1) \cup \gamma_1^{-1}[N_1] \cup (N_2 \cap \mathcal{N}'_2)\,. \tag{2}$$

From Equation (2) we see that $N_1 \mid_\gamma N_2$ is the union $N_1 \cup N_2$ but with the parts of N_1 and N_2 that are identified via γ_1 and γ_2 replaced by the shared names $\gamma_1^{-1}[N_1] = \gamma_2^{-1}[N_2]$. Also, for a constraint g, we write $\gamma(g)$ for the formula obtained by replacing port names in $\gamma_1[\mathcal{N}] \subseteq \mathcal{N}_1$ and $\gamma_2[\mathcal{N}] \subseteq \mathcal{N}_2$ by the corresponding name in $\mathcal{N}$.

Definition 4.2. For two constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with port synchronization function γ, the constraint automaton $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2$, called the γ-synchronous product of $\mathcal{A}_1$ and $\mathcal{A}_2$, is given by $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2 = (S_1 \times S_2, \mathcal{N}', \rightarrow, \langle s_0^1, s_0^2 \rangle)$ where $\mathcal{N}' = \mathcal{N}_1 \mid_\gamma \mathcal{N}_2$ and the transition relation $\rightarrow$ is determined by the rules

$$\frac{s_1 \xrightarrow{N_1, g_1}_1 t_1 \qquad N_1 \subseteq \mathcal{N}'_1}{\langle s_1, s_2 \rangle \xrightarrow{N_1, g_1} \langle t_1, s_2 \rangle} \qquad \frac{s_2 \xrightarrow{N_2, g_2}_1 t_2 \qquad N_2 \subseteq \mathcal{N}'_2}{\langle s_1, s_2 \rangle \xrightarrow{N_2, g_2} \langle s_1, t_2 \rangle}$$

and

$$\frac{s_1 \xrightarrow{N_1, g_1}_1 t_1 \qquad s_2 \xrightarrow{N_2, g_2}_2 t_2 \qquad \gamma_1^{-1}(N_1) = \gamma_2^{-1}(N_2)}{\langle s_1, s_2 \rangle \xrightarrow{N_1 \mid_\gamma N_2, \gamma(g_1 \wedge g_2)} \langle t_1, t_2 \rangle} \tag{3}$$

In the above setting, for a port $n \in \mathcal{N}$, the idea is that the ports $n_1 = \gamma_1(n) \in \mathcal{N}_1$ and $n_2 = \gamma_2(n) \in \mathcal{N}_2$ synchronize. Thus, either n_1 and n_2 have both flow or n_1 and n_2 have both no flow, expressed as n having flow or no flow, respectively. The resulting automaton, the so-called synchronized product automaton $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2$, follows the flow of $\mathcal{A}_1$ and A_2, based on the first two rules for the transition relation, but requires the flow on its ports in $\mathcal{N}$ to be agreed upon by $\mathcal{A}_1$ and $\mathcal{A}_2$.

A parallel composition $P_1 \parallel P_2$ in mCRL2, being based on the process algebra ACP [BBR10], has its transitions derived from the steps of its left component P_1, its right component P_2, or their synchronization:

$$P_1 \parallel P_2 \;=\; P_1 \between P_2 + P_2 \between P_1 + P_1 \mid P_2$$

Here, $\between$ denotes the left merge, an auxiliary operator in mCRL2. Given two processes p and q, the left merge, written $p \between q$, requires p to execute an action first and thereafter continues as the parallel composition of the remainder of p and q. The summand $P_1 \mid P_2$ expresses the synchronization of P_1 and P_2, requiring an multiaction of P_1 to synchronize with an multiaction of P_2 as specified by the synchronization function γ. For example, $(a \cdot Q) \mid (b \cdot Q')$ yields $c \cdot (Q \parallel Q')$ if $\gamma(a, b) = c$. For mCRL2 processes, if not stated otherwise, the communication operator is assumed to yield the inaction δ. In case data is considered, an action may carry an data item that flows through the port associated with the action. Synchronization requires the data of the synchronizing ports to be equal. Thus, we have that $(A(d) \cdot Q) \mid (B(d) \cdot Q')$ yields $C(d) \cdot (Q \parallel Q')$, but $(A(d_1) \cdot Q) | (B(d_2) \cdot Q')$, for different data items d_1, d_2 yields the deadlock process δ, which vanishes in an alternative or parallel composition. A similar refinement can be applied for coloring.

In the context of the synchronization product $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2$ of constraint automata two aspects are important. Firstly, the port synchronization function γ decides which ports synchronize, hence which sets of port names are non-trivially combined. Secondly, ports that are to be synchronized, i.e. $\gamma_1[\mathcal{N}] \subseteq \mathcal{N}_1$ and $\gamma_2[\mathcal{N}] \subseteq$

$\mathcal{N}_2$, have their names erased from the alphabet. Thus, for the first issue, we define the attributes of the mCRL2 communication operator for a parallel composition as determined by a port synchronization for the synchronized product of two constraint automata. For the second issue, we will apply a specific blocking operator determined by the port synchronization.

Definition 4.3. Given a port synchronization function γ with mappings $\gamma_1 \colon \mathcal{N} \to \mathcal{N}_1$, $\gamma_2 \colon \mathcal{N} \to \mathcal{N}_2$, the mCRL2 process $P_1 \parallel_\gamma P_2$, called the γ-synchronized merge of P_1 and P_2, is defined as follows:

$$P_1 \parallel_\gamma P_2 \;=\; \partial_B(\Gamma_C(P_1 \parallel P_2))$$

where $B = \gamma_1[\mathcal{N}] \cup \gamma_2[\mathcal{N}]$ is the set of blocked actions and $C = \{\, \gamma_1(n)|\gamma_2(n) \mapsto n \mid n \in \mathcal{N} \,\}$ is the set of communications.

We are now in a position to formulate a soundness result for our translation with respect to the parallel composition combined with appropriate synchronization. Theorem 4.1 states for the data-agnostic case that the mCRL2 process associated to a synchronized product of two constraint automata is the same as the synchronized merge of the mCRL2 processes corresponding to the two individual constraint automata. For brevity, we restrict to the case of a binary product. However, the result straightforwardly generalizes to an arbitrary number of constituents. Here, equality is modulo strong bisimulation [Mil89], notation $\leftrightarroweq$. It is noted that from a logical point of view, branching bisimilar processes, hence also strongly bisimilar processes, can be used interchangeably within the mCRL2 toolset.

Theorem 4.1. Let $\mathcal{A}_1 = (S_1, \mathcal{N}_1, \to_1, s_0^1)$ and $\mathcal{A}_2 = (S_2, \mathcal{N}_2, \to_2, s_0^2)$ be two constraint automata with disjoint sets of port names and let γ be a port synchronization for $\mathcal{A}_1$ and $\mathcal{A}_2$. Then it holds that

$$\mathit{proc}(\mathcal{A}_1, s_1) \parallel_\gamma \mathit{proc}(\mathcal{A}_2, s_2) \;\leftrightarroweq\; \mathit{proc}(\mathcal{A}_1 \bowtie_\gamma A_2, \langle s_1, s_2 \rangle)\,.$$

Proof. We verify, by checking the usual transfer conditions, that the relation

$$\mathcal{R} = \{\, (\,\mathit{proc}(\mathcal{A}_1, s_1) \parallel_\gamma \mathit{proc}(A_2, s_2),\, \mathit{proc}(\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2, \langle s_1, s_2 \rangle)\,) \mid s_1 \in \mathcal{A}_1,\, s_2 \in \mathcal{A}_2 \,\}$$

is a strong bisimulation relation. For brevity we write $\mathit{proc}_i(s_i)$ for $\mathit{proc}(\mathcal{A}_i, s_i)$, $i = 1, 2$, and $\mathit{proc}(s_1, s_2)$ for $\mathit{proc}(\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2, \langle s_1, s_2 \rangle)$ and suppress data constraints from the labels.

Suppose $\mathit{proc}_1(s_1) \parallel_\gamma \mathit{proc}_2(s_2) \xrightarrow{N} P$. From the semantics of $\parallel_\gamma$ we obtain

(i) $\exists P_1 \colon \mathit{proc}_1(s_1) \xrightarrow{N} P_1$, $N \subseteq \mathcal{N}_1'$, and $P = P_1 \parallel_\gamma \mathit{proc}_2(s_2)$,

(ii) $\exists P_2 \colon \mathit{proc}_2(s_2) \xrightarrow{N} P_2$, $N \subseteq \mathcal{N}_2'$, and $P = \mathit{proc}_1(s_1) \parallel_\gamma P_2$, or,

(iii) $\exists P_1, P_2, N_1, N_2 \colon \mathit{proc}_1(s_1) \xrightarrow{N_1}_1 P_1$, $\mathit{proc}_2(s_2) \xrightarrow{N_2}_2 P_2$, $N = N_1 \mid_\gamma N_2$ and $P = P_1 \parallel_\gamma P_2$.

By the definition of $\mathit{proc}_1(s_1)$ and $\mathit{proc}_2(s_2)$ we then have

(i) $\exists t_1 \colon s_1 \xrightarrow{N}_1 t_1$, $N \subseteq \mathcal{N}_1'$, and $P = \mathit{proc}_1(t_1) \parallel_\gamma \mathit{proc}_2(s_2)$,

(ii) $\exists t_2 \colon s_2 \xrightarrow{N}_2 t_2$, $N \subseteq \mathcal{N}_2'$, and $P = \mathit{proc}_1(s_1) \parallel_\gamma \mathit{proc}_2(t_2)$, or,

(iii) $\exists t_1, t_2, N_1, N_2 \colon s_1 \xrightarrow{N_1}_1 t_1$, $s_2 \xrightarrow{N_2}_2 t_2$, $N = N_1 \mid_\gamma N_2$ and $P = \mathit{proc}_1(t_1) \parallel_\gamma \mathit{proc}_2(t_2)$.

Therefore, we have $\langle s_1, s_2 \rangle \xrightarrow{N} \langle t_1, s_2 \rangle$ in $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2$, thus $\mathit{proc}(s_1, s_2) \xrightarrow{N} \mathit{proc}(t_1, s_2)$, while, in view of property (i), $\mathit{proc}_1(t_1) \parallel_\gamma \mathit{proc}_2(s_2)\ \mathcal{R}\ \mathit{proc}(t_1, s_2)$. A symmetric remark applies regarding (ii). Finally, $\langle s_1, s_2 \rangle \xrightarrow{N} \langle t_1, t_2 \rangle$ and $\mathit{proc}(s_1, s_2) \xrightarrow{N} \mathit{proc}(t_1, t_2)$, while $\mathit{proc}_1(t_1) \parallel_\gamma \mathit{proc}_2(t_2)\ \mathcal{R}\ \mathit{proc}(t_1, t_2)$ regarding (iii).

Now suppose $\mathit{proc}(s_1, s_2) \xrightarrow{N} P$. By definition of $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2$, either

(i) $N \subseteq \mathcal{N}_1'$ and $\exists t_1 \colon \langle s_1, s_2 \rangle \xrightarrow{N} \langle t_1, s_2 \rangle$ based on $s_1 \xrightarrow{N}_1 t_1$, and $P = \mathit{proc}(t_1, s_2)$,

(ii) $N \subseteq \mathcal{N}_2'$ and $\exists t_2 \colon \langle s_1, s_2 \rangle \xrightarrow{N} \langle s_1, t_2 \rangle$ based on $s_2 \xrightarrow{N}_2 t_2$, and $P = \mathit{proc}(s_1, t_2)$, or,

(iii) $N \not\subseteq \mathcal{N}_1'$, $N \not\subseteq \mathcal{N}_2'$ and $\exists t_1, t_2, N_1, N_2 \colon \langle s_1, s_2 \rangle \xrightarrow{N} \langle t_1, t_2 \rangle$ based on $s_1 \xrightarrow{N_1}_1 t_1$ and $s_2 \xrightarrow{N_2}_2 t_2$ with $N = N_1 \mid_\gamma N_2$ and $P = \mathit{proc}(t_1, t_2)$.

Then, we have $\mathit{proc}_1(s_1) \parallel_\gamma \mathit{proc}_2(s_2) \xrightarrow{N} \mathit{proc}_1(t_1) \parallel_\gamma \mathit{proc}_2(s_2)$ and $\mathit{proc}_1(t_1) \parallel_\gamma \mathit{proc}_2(s_2)\ \mathcal{R}\ \mathit{proc}(t_1, s_2)$ in

case of (i), a symmetric observation in case of (ii), and $proc_1(s_1) \parallel_\gamma proc_2(s_2) \stackrel{N}{\rightarrow} proc_1(t_1) \parallel_\gamma proc_2(t_2)$ and $proc_1(t_1) \parallel_\gamma proc_2(t_2)\ \mathcal{R}\ proc(t_1, t_2)$ in case of (iii).

Conclusion, $\mathcal{R}$ is a strong bisimulation relation and, for all $s_1 \in \mathcal{A}_1$, $s_2 \in \mathcal{A}_2$, it holds that $proc(\mathcal{A}_1, s_1) \parallel_\gamma proc(\mathcal{A}_2, s_2) \leftrightarrows proc(\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2, \langle s_1, s_2 \rangle)$. □

The theorem above goes beyond the order-independence result of Section 3. Theorem 3.1 relates to the static structure of the Reo connector, whereas Theorem 4.1 based on the constraint automaton of the connector relates to its dynamics. Cornerstone of constraint automaton-based semantics of Reo is the fact that the semantics of a connector is independent of the way it is considered to be built up from its constituting channels. Theorem 4.1 implies that this is the case for our encoding into `mCRL2` too. Also note that the above theorem, in the present setting of a parallel construct involving a port synchronization function, shows strong bisimilarity, whereas the original result of [BSAR06] claims, for general composition, language equivalence with respect to timed data streams.

In order to generalize the result to the case of data-aware Reo connectors, we have to provide a counterpart of the γ-synchronous product of two constraint automata that takes the passing of data into account. The crucial transition is given by rule (3). Here, the data constraint $\gamma(g_1 \wedge g_2)$, i.e. the conjunction of g_1 and g_2 but replacing the port n_1 by n in g_1 and n_2 by n in g_2 in case $\gamma(n) = (n_1, n_2)$, imposes a number of restrictions on the data flowing through the ports in $N_1 \mid_\gamma N_2$. For the primitive channels, the CA-description given by Figure 2 and the data-aware encoding in `mCRL2` as given by Table 2 coincide, as can be straightforwardly obtained from the transitional semantics of the two formalisms. Synchronization of the relevant channel ends meeting at a node, viz. one selected source end and all sinks, implies equality of the data that is transferred. Thus now for the communication operator Γ_C as occurring in the γ-synchronized merge of Definition 4.3 we read $C = \{\, \gamma_1(n)(d) \mid \gamma_2(n)(d) \mapsto n(d) \mid n \in \mathcal{N},\ d \in Data \,\}$. The proof of bisimilarity of $proc(\mathcal{A}_1, s_1) \parallel_\gamma proc(\mathcal{A}_2, s_2)$ and $proc(\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2), \langle s_1, s_2 \rangle)$ follows the lines of Theorem 4.1 with exactly the same bisimulation relation $\mathcal{R}$, but with a refined case analysis. The formal proof requires a detailed definition of the constraint syntax, a direction we do not pursue here.

Along the same lines correspondence results for the other encodings can be obtained. For example, in the case of the coloring semantics, the communication operator involves set of the form

$$\{\, \gamma_1(n(\mathit{flowcolor}(d)) \mid \gamma_2(n(\mathit{flowcolor}(d)) \mapsto n(\mathit{flowcolor}(d)) \mid \\ n \in \mathcal{N},\ d \in Data,\ \mathit{flowcolor} \in \{\, \mathit{flow}, \mathit{noflowG}, \mathit{noflowR} \,\} \,\}$$

to express that agreement on the *flowcolor* is required between node and the channels incident upon it, no matter the channel is part of the one or the other subconnector. For timed Reo connectors and action constraint automata the situation is simpler. In the constraint automata time is not present in constraints, while in the `mCRL2` representation time is not passed as a parameter. For the action constraint automata model a port together with its action type constitute the `mCRL2` action to synchronize. Apart from this, Theorem lemma-product goes through without change.

5. Case Study

In this section, we illustrate our way of formal modelling of Reo in the setting of service-based processes. As a running example, we consider the Access Control System (ACS) introduced in [KH06]. Figure 10 shows the UML activity diagram for this process as developed by Kraemer and Herrmann. The system is used to restrict the opening of a door to a group of trusted people which have to authenticate themselves with an identity card and a personal identification number. This identification data (referred to as *pId*) is provided at a panel next to the door. The panel and the door opening mechanisms are handled by a local control station installed close to the door. The local station is connected with a central station calling both an authentication and an authorization service which check the *pId* by accessing corresponding databases.

The Reo model of this process is shown in Figure 11. Such a model can be obtained automatically from the initial UML activity diagram by one-by-one mapping the nodes and flow arrows to Reo nodes and channels. Thus, one can observe that fork and merge nodes in UML correspond to simple (replicate and merge) nodes in Reo, a decision node in UML corresponds to an exclusive router node (a shorthand for the exclusive router circuit) in Reo while a join node in UML is represented by a Reo SyncDrain channel. In the latter case, the subsequent outgoing channel is connected to the end of the SyncDrain whose data item is propagated further in the circuit. In the case of dataflow modeling, i.e., assuming that data values provided and consumed by

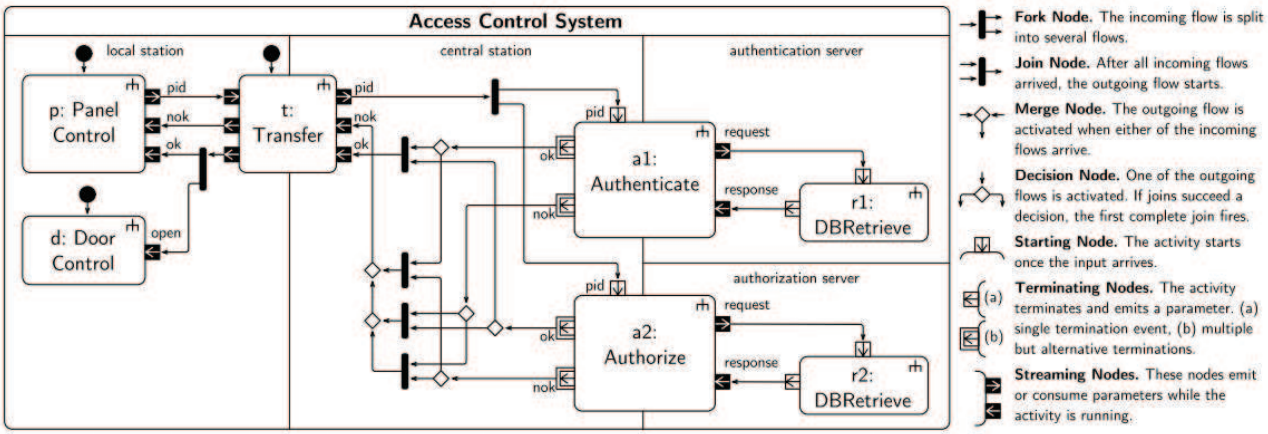


Fig. 10. Access Control System: UML model

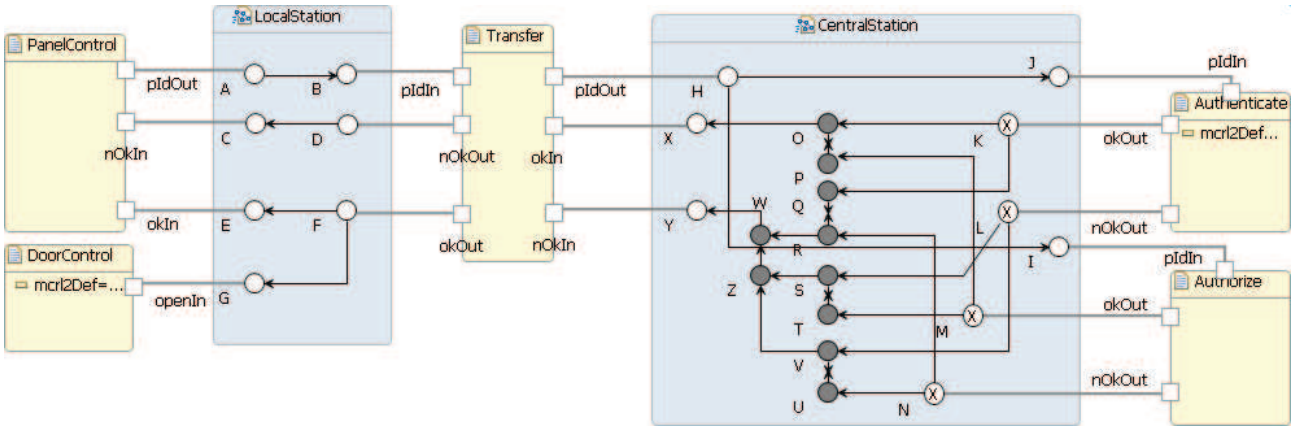


Fig. 11. Access Control System: Reo model

components do matter, this capability is important as it allows us to select which value will be sent to the input ports of the *Transfer* component, *ok* or *nOk*. The initial UML activity diagram represents a simple control flow and this aspect is neglected. In this picture, for the sake of simplicity, we omit modeling of the communication with databases. We can assume that the decision about the authentication and authorization of a given person is an internal logic of the *Authenticate* and *Authorize* services and that database calls are not observable by external users.

Usually, internal behavior of a service is not known and services are composed based on their observable behavioral interfaces, e.g., given in a form of input/output (I/O) automata. An I/O automaton is a labeled transition system with actions classified as input, output and internal actions. For simple stateless services such a model can be obtained automatically from their WSDL specifications. In our example, we assume that observable component behavior is essentially an I/O automaton where actions are parameterized with acceptable data input. We can specify the behavior of our components in `mCRL2` as follows:

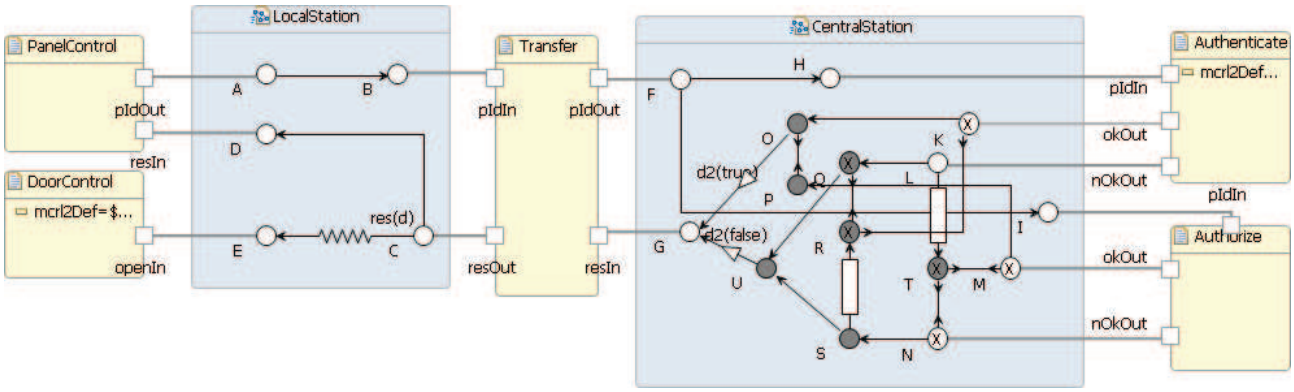


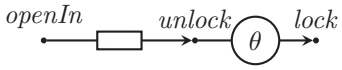
Fig. 12. Optimized Access Control System: Reo model

$$
\begin{aligned}
\mathsf{PanelControl} &= \Sigma_{pId:Nat}\, pIdOut(pId) \cdot (okIn(ok) + nOkIn(nOk)) \cdot \mathsf{PanelControl}\,; \\
\mathsf{DoorControl} &= \Sigma_{d:Data}\, openIn(d) \cdot \mathsf{DoorControl}\,; \\
\mathsf{Transfer} &= \Sigma_{pId:Pos}\, pIdIn(pId) \cdot pIdOut(pId) \cdot \\
&\qquad (okIn(ok) \cdot okOut(ok) + nOkIn(nOk) \cdot nOkOut(nOk)) \cdot \mathsf{Transfer}\,; \\
\mathsf{Authenticate} &= \Sigma_{pId:Pos}\, pIdIn(pId) \cdot ((pId \in \mathcal{P}_1) \rightarrow okOut(ok) \diamond nOkOut(nOk)) \cdot \mathsf{Authenticate}\,; \\
\mathsf{Authorize} &= \Sigma_{pId:Pos}\, pIdIn(pId) \cdot ((pId \in \mathcal{P}_2) \rightarrow okOut(ok) \diamond nOkOut(nOk)) \cdot \mathsf{Authorize};
\end{aligned}
$$

Here $\mathcal{P}_1$ and $\mathcal{P}_2$ are sets of personnel permitted to enter the door for the *Authenticate* and *Authorize* services, respectively. In our specification, a personal identification number is represented as a data item of type *Pos* (positive integer) while the decision to grant or refuse access control is propagated through corresponding ports using signals denoted as *ok* or *nOk*. To denote that the door is unlocked when some data item of unspecified data type arrives to the port *openIn*, we refer to a global type *Data*. This type is defined by the scope of the connector and is obtained as a conjunction of all data types that can enter the boundary nodes of the connector. In our case it can be defined as

sort $Data = \mathbf{struct} d_1(pId{:}Pos) \mid ok \mid nOk\,;$

In the component specifications, we deliberately omit internal actions such as, e.g., *unlock* and *lock* actions for the *DoorControl* component that represent the temporal unlocking and permanent locking of the door. Note that although the primary goal of Reo is to model component interaction, it can be successfully used for the specification of internal behavior of components as well. Thus, the internal behavior of the *DoorControl* component can be modeled by a Reo circuit composed of a FIFO and a Timer channel:



Given such a model and assuming that *lock* and *unlock* nodes are internal Reo nodes, the above specification for the *DoorControl* in `mCRL2` can be inferred automatically. In the case of timed mapping, the `mCRL2` *at-operator* '@' can be used to define that the door is locked again after being open for θ time units:

$\mathsf{DoorControl} = \Sigma_{d:Data,t:Real}\, openIn(d) \cdot lock@t \cdot unlock@(t+\theta) \cdot \mathsf{DoorControl}\,;$

The semantics of the ACS model specified in Reo with I/O data annotations is obtained from the generated `mCRL2` code. Assuming that the model has a finite state space, we can use `mcrl22lps`, `lps2lts` and `ltsgraph` tools to generate and visualize a parametrized LTS illustrating the behavior of our system specification. For example, Figure 13 shows a state space for the ACS where the admissible input data is restricted to $pId \leq 2$ and access is granted to a person with $pId = 1$. This LTS shows dataflow through the boundary ports of Reo connectors (white nodes in Figure 11). Notice that after the data value $d_1(1)$ on port A linked to the port *pIdOut* of the *PanelControl* has been observed, eventually, a data item *ok* is received on ports E and G

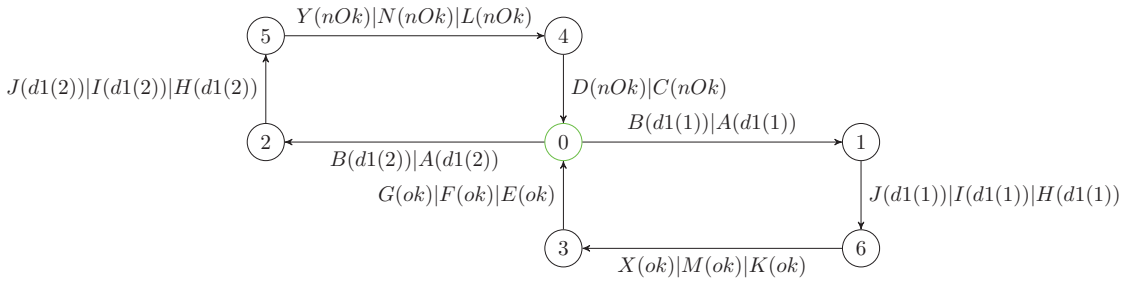


Fig. 13. Access Control System: LTS semantics

corresponding to ports *okIn* of the *PanelControl* and port *openIn* of the *DoorControl*. Similarly, for the $pId = 2$, a value *nOk* is received on the port *D* related to the port *nOkIn* of the *PanelControl*.

The ACS system does not open the door if either of the *Authenticate* and *Authorize* services returns a negative authorization result. However, the system waits for both of these services to provide some result even in the case when one service already denied the access for a given *pId*. Imagine now that we want to design a circuit that instead immediately sends a negative result to the user whenever any of the two access control services yields a negative result. To make it even more challenging and to illustrate the ability of Reo to bridge a gap between mismatching service interfaces, let us assume slightly different specifications of the components involved. Thus, we suppose that the *PanelControl* and *Transfer* components represent the result of personal identification as a data element $res : Bool$, while *Authenticate* and *Authorize* services dispense positive and negative results *ok* and *nOk* of authorization and authentication to ports *okOut* and *nOkOut*, respectively, as in the initial model. The `mCRL2` specifications for the *PanelControl* and *Transfer* change to:

$$\begin{aligned} \mathsf{PanelControl} &= \Sigma_{pId:Pos,res:Bool}\ pIdOut(pId) \cdot resIn(res) \cdot \mathsf{PanelControl}\,; \\ \mathsf{Transfer} &= \Sigma_{pId:Pos,res:Bool}\ pIdIn(pId) \cdot pIdOut(pId) \cdot resIn(res) \cdot resOut(res) \cdot \mathsf{Transfer}\,; \end{aligned}$$

while the global data sort for this model is defined as

$$\textbf{sort}\ Data = \textbf{struct}\ d_1(pId{:}Pos) \mid d_2(res{:}Bool) \mid ok \mid nOk\,;$$

Figure 12 shows an optimized ACS where the *nOk* signal dispensed is first sent to the door and panel controllers without synchronizing with the second signal. Two pairs of connected FIFO and SyncDrain channels are used to remember that the *nOk* signal was dispensed and clean up the circuit when the second component yields the result, either *ok* or *nOk*. Two transformer channels are used to set up the value of the boolean variable *res* to *true* or *false* after receiving either *ok* or *nOk* signals, respectively.

The state space of the optimized ACS for $pId \leq 2$ is shown in Figure 14. For the input $pId = 1$ the ACS behaves as before, while in the case of $pId = 2$ many other states appear because of the absence of synchronization between the *nOkOut* ports of *Authenticate* and *Authorize* services. Since the visual analysis of the system by humans is not possible anymore, we should rely on model checking tools to verify essential properties of the ACS. Thus, to ensure that for any person with $pId \neq 1$ the access will be denied the following μ-calculus formula can be used

$$\begin{aligned} &\exists\, pId{:}Pos\,.\, val(pId \neq 1)\ \wedge \\ &\quad [\,true^*.\, A(d_1(pId))|B(d_1(pId))\,]\,\mu X \cdot \big(\,[\,\neg(\,C(d_2(false))|D(d_2(false))\,)\,]\,X\ \wedge\ \langle true\rangle\, true\,\big) \end{aligned}$$

In this formula, the boolean function $val(expr{:}Bool)$, provided in the `mCRL2` property specification language, is used to evaluate a boolean condition that restricts the domain of the data parameter. For both versions of the ACS this property holds. Similarly, we can check that for every request with $pId \in \mathcal{P}_1 \cap \mathcal{P}_2$ the access is granted. Note that in `mCRL2` actions a and $a|b$ are considered different and a formula that requires an action a to occur will not be satisfied if this action occurs solely within a multiaction. Practically this problem can be solved by hiding irrelevant actions in the model. If this is not possible, one has to explicitly specify all multiactions containing a in the formula. In this respect it would be helpful if the `mCRL2` property

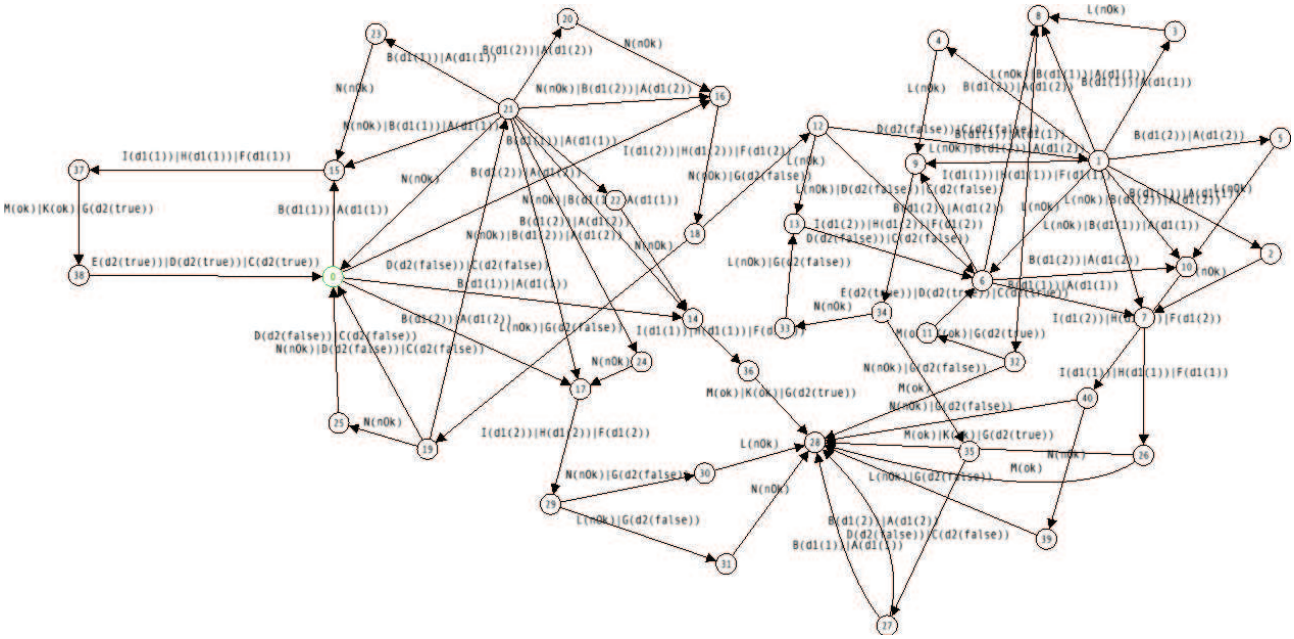


Fig. 14. Optimized Access Control System: LTS semantics

specification format is extended with the ability to define formulae that hold when some action a occurs either independently or as part of a multiaction.

We can further modify the initial model of the ACS by introducing a LossyFIFO buffer to temporary store a request that has arrived while the identification services are not responding or still recovering from failure. Any following request will be accepted by the system and ignored. For the analysis of this model, a context-sensitive mapping should be used. Furthermore, if we want to estimate the end-to-end ACS response time, we can associate time delays with each channel and use the ACA-based mapping. The LTS produced by the mCRL2 tool in this case will show the order of data transfer in synchronous parts of the circuit which currently are represented by a single transition with multiactions, e.g., $H(x)|I(x)|J(x)$ in Figure 13. Given such a model, a Q-algebra can be used to compute the total delay of a synchronous transaction, cf. [CK07]. Thus, assuming that the time delays to transfer a request to the *Authenticate* service through the channel $\mathsf{Sync}(H, I)$ and to the *Authorize* service though the channel $\mathsf{Sync}(H, J)$ equal t_1 and t_2, respectively, the total time delay for the transition $H(x)|I(x)|J(x)$ can be computed as $max(t_1, t_2)$. Currently, the mCRL2 toolset does not provide tools for performance evaluation and, therefore we do not include QoS parameters of Reo channels to our encoding. However, our future work includes the provisioning of tools for Reo performance evaluation and we consider the incorporation of various QoS metrics to the LTS produced by our mCRL2 generation plug-in.

6. Tool Support

We implemented the conversion from Reo to mCRL2 discussed above as an extension to the Extensible Coordination Tools (ECT), see [AKM+08, Kra11]. ECT is a framework for modeling, verification and execution of component-based and service-oriented systems. It consists of a set of integrated tools for the Eclipse platform.[5] The framework provides functionality for converting high-level modeling languages, such as UML, BPMN and BPEL to Reo [CKA10], for editing and animation of Reo models, generation of automata-based semantical models from Reo, modeling and verification of QoS properties and tight integrations with external model checking tools such as Vereofy [BBKK09] or PRISM [KNP02].

Our conversion tool, shown in the bottom part of Figure 15, takes as input a Reo circuit and generates

[5] http://www.eclipse.org

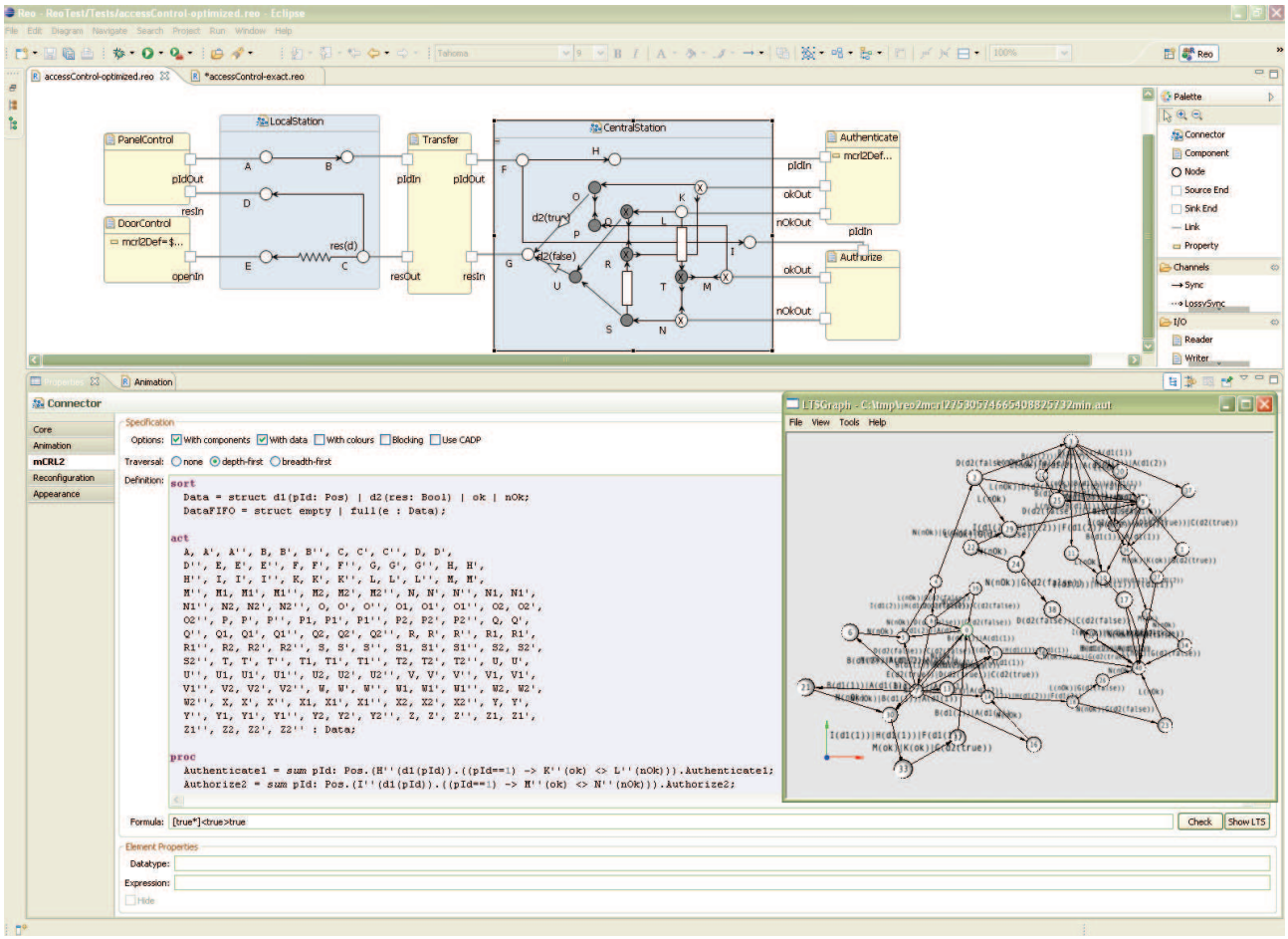


Fig. 15. Extensible Coordination Tools: Reo to mCRL2 plug-in

automatically a corresponding mCRL2 specification. The tool is tightly integrated with the graphical Reo editor, depicted in the upper part of Figure 15. instance, the tool allows to incorporate process definitions for the components attached at the boundary of a connector. Moreover, the user can choose between a data-agnostic and a data-aware encoding, both based on the CA semantics (cf. Section 2.1). Furthermore, the tool supports a context-sensitive encoding based on the coloring semantics (cf. Section 2.3) and a translation according to the ACA semantics (cf. Section 2.4), which allows to inspect data flow events within synchronous regions of a connector. Additionally, data types of components or services coordinated by Reo, as well as data constraints for data dependent channels such as the Filter or Transform channel can be defined in the tool. Technically, this additional information is saved as annotations in the Reo model and is automatically merged when generating the final mCRL2 specification. This way we can ensure that the mCRL2 code can be regenerated at any point without manual changes.

The tool further includes an integration with mCRL2's model checking and state space visualization tools. In particular, we use the mcrl22lps tool for generating linear process specifications (LPS) from mCRL2 code, lps2lts and ltsconvert for generating and minimizing labeled transitions systems, lps2pbes for model checking formulas specified in modal μ-calculus, and finally ltsgraph for visualizing state spaces.

Reo connectors can be converted to components and reused in higher-level process models. In this case, the semantics of the component will be inferred from the connector. Alternatively, a user can add an arbitrary component to the model as we illustrated in our case study in Section 5. For some frequently used components, e.g., *Variable* [KKdV10], the system adds the mCRL2 specifications automatically. For other components such a behavior can be specified manually or inferred from the associated external specification, e.g. WSDL. The

similar principle is applied to customized channels. A user can add a new component or a channel and define their behavior using mCRL2. A simple template language is used to generate an mCRL2 specification. For example,

$$\$proc \;=\; \Sigma_{d:Data}\, \$inPort(d)\,.\,\$outPort(d)\,.\,\$proc\,;$$

defines a component or custom channel with the behavior of a FIFO channel. Here $\$proc$ is substituted with a unique process name for the modeled component. If a channel or a component is supposed to deal with a global data type generated for the overall process model, such a data type can be specified using the variable $\$data$ which will be substituted with the generated data type name by the conversion tool. This way, custom channels and components can be specified individually, without the need to manually adapt the final mCRL2 specification.

Furthermore, our tool provides an integration with the CADP toolset.[6] CADP stands for the Construction and Analysis of Distributed Processes and comprises a toolbox for the design of communication protocols and distributed systems. CADP offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking and has been used in many industrial projects. In a nutshell, the toolset provides the following functionalities:

- Compilers for several input formalisms (e.g. LOTOS, finite state machines running in parallel and synchronized together using process algebra operators or synchronization vectors).
- Several equivalence checking tools (minimization and comparisons modulo bisimulation relations), such as BCG_MIN and BISIMULATOR.
- Several model-checkers for various temporal logics and μ-calculus, such as EVALUATOR and XTL.
- Several verification algorithms combined together: enumerative verification, on-the-fly verification, symbolic verification, compositional minimization, partial order reduction, distributed model checking, etc.
- Advanced tools for process visual inspection and performance evaluation.

To obtain a specification for a given Reo model in a format processable by the CADP tools, we convert LPS to LTS specifications and save them in the Aldebaran format. This LTS then is converted to the BCG format using the CADP tool bcg_io. We integrated this approach together with the EVALUATOR model checker in our conversion tool in ECT. The main advantage of using model checking tools from the CADP suite is the ability to extract counterexamples which is currently only indirectly supported for the lps2pbes model checker from the mCRL2 toolbox.

In our encoding of Reo in mCRL2 every primitive (channels, nodes and components) is translated to a separate process, all of which are then run in parallel. Every primitive end corresponds to an action in this setting. Therefore, the derived specifications usually consist of a rather large number of processes and an even larger number of actions. However, the interaction among all these processes is rather local, e.g. a channel communicates only with its source and target nodes. Consequently, we observe a major performance loss in the generation of a linear process specification with the mcrl22lps tool, caused by a state space explosion. To overcome this problem, we can construct the connector in a stepwise fashion by adding processes one by one and immediately applying the composition and encapsulation operators. We have explained this principle already in Section 3.

Using this adaption of the original approach, the intermediate state spaces remain relatively small and mcrl22lps can process them more efficiently. Furthermore, we can use the topology of the connector to determine an order for the processes that significantly improves the runtime of the linearization algorithm. In our experiments, a depth-first traversal on the connector graph yielded the best results. In the following we give a benchmark that shows the effect of our optimizations. Table 4 summarizes the runtimes of two example connectors:

(a) n parallel FIFOs, the source ends of all of which are connected to a common exclusive router node, with their sink ends all connected to a common merge node. This connector implements an unordered buffer (a bag) of size n.
(b) n FIFOs in a sequence, connected using normal nodes. This connector implements a FIFO-n.

[6] http://www.inrialpes.fr/vasy/cadp

Table 4. runtimes of the `mcrl22lps` tool in seconds

(a) parallel FIFOs

n	naive	BFS	DFS
4	0.73	0.09	0.07
5	12.89	0.20	0.08
6	408.61	0.66	0.11
7		3.20	0.14
8		21.67	0.19
9		229.80	0.27
10			0.37
⋮			⋮
28			22.38
29			26.57
30			30.99

(b) sequential FIFOs

n	naive	BFS/DFS
4	1.22	0.07
5	71.28	0.08
6		0.12
7		0.18
8		0.26
9		0.40
10		0.71
11		1.24
12		2.30
13		4.52
14		9.31
15		20.09

The benchmarks were performed on a desktop computer with a 2.40GHz quad-core CPU and 8GB memory, running Linux 2.6.27 and the development version of `mCRL2` (revision 8013). The two tables show the number of FIFOs and the runtimes of `mcrl22lps` in seconds. The columns show the results, respectively, for the naive encoding and two stepwise constructions based on a breadth-first search (BFS) and a depth-first search (DFS) on the connector graph (BFS and DFS separate for the parallel FIFO, BFS and DFS combined for the sequential FIFO). Empty cells mean that the computation took more than ten minutes. The example of n parallel FIFOs (Table 4a) shows that a breadth-first search can already improve the runtime. However, the growth is still exponential by a factor of 10 approximately, whereas the depth-first traversal yields much better results by a factor of around 1.2. In the sequential FIFO example (Table 4b), DFS and BFS coincide and have a factor of about 2. Note that in (b) the state space in fact grows exponentially, whereas in (a) it is linear in the number of FIFOs.

The `mCRL2` converter in ECT provides a powerful means for verifying data-, context- and time-dependent behavior. To the best of our knowledge there exists no other modeling tool that provide this functionality for verifying connector and workflow behavior to such extent.

7. Related Work

The absence of formal semantics in many business process modeling notations and specification languages poses the need of their transformations into models having a more formal semantical basis such as process algebras, finite-state machines, Petri nets, or such as Reo, as proposed here. A survey in [HCLY07] compares a set of academic and commercial products for service-based process modeling, namely, Bind Studio, BPEL Orchestration Server, WebDesigner, eFlow, SWORD, DAML-S, WCPT, SODA, and WebDG. Most of these tools do not provide means for formal process verification, while others (SWORD and DAML-S) rely on Petri nets. COSA and Protos are two popular workflow modelers that permit both free specifications of business processes to support initial and conceptual modeling and associate formal Petri nets-based semantics to these models for automated process validation.

Petri nets are a powerful model for concurrent process specification supported by a number of industrial tools [Aal98]. Among the most well known verification tools for Petri nets are Woflan [Aal99] and CPN Tools [Rea03]. Woflan has support for the verification of logical correctness of workflow nets, a restricted version of Petri nets for process modeling. CPN Tools represent a toolkit for editing, simulating and analyzing colored Petri nets where tokens are typed and can be used to distinguish several data elements.

Classical Petri nets are often not sufficient for advanced workflow modeling. To express certain behavioral aspects, e.g., process cancellation and/or compensation, Petri nets are usually extended with additional elements. Yasper [HvPS05] is a modeling framework that supports Petri nets extended with inhibitor and reset arcs. As has been pointed out by Raedts et al. [RPU$^+$07], this significantly reduces the number of software tools able to analyze such models. Similarly to our work, this paper uses the `mCRL2` toolset for analyzing Yasper models. However, we do not limit our approach to structural verification of workflow models, but target at time- and data-aware analysis, while Raedts et al.'s translation represents only the number of

data tokens in Petri-nets, but not their content. For dataflow modeling, Hidders et al. [HKS+05, HKS+08] extend Petri nets with nested relational calculus, a database query language over composite data types. The authors introduce a set of refinement rules which, being applied hierarchically, guarantee that the final workflow net is sound, i.e., initiated and terminated with a single token in the input and output nodes, all input data is processed, no 'debris data' is left behind and the output is always eventually computed. Similar to this approach, we deal with concrete data items rather than abstract tokens. However, [HKS+05, HKS+08] do not provide integration with model checking tools for verification of the constructed dataflow models. Trčka et al. [TAS08] consider workflow correctness criteria analyzing Petri net-based models extended with 'read', 'write' and 'destroy' operations to enable mixed control- and dataflow analysis. This work is conceptually close to ours. Reader, writer and custom components in our framework perform the same functions as read and write operations in this work, while the loss of data by lossy or synchronous drain channels is similar to their destruction. The authors provide a set of CTL* properties to identify some common workflow errors. Being based on μ-calculus, the `mCRL2` property specification format subsumes CTL*, and, therefore the corresponding properties can be stated and verified in our framework as well. Additionally, `mCRL2` provides means for further data manipulation not covered by [TAS08].

TINA [BRV04] and Roméo [LRST09] are frameworks for the verification of timed Petri nets. TINA provides an editor for graphically or textually described Petri nets, construction of reachability graphs, structural analysis, state/event LTL model checker and path analysis for time Petri nets. Roméo is a toolset that provides a graphical user interface to edit and design time Petri nets and modules for translating workflow models to timed and stopwatch automata, graphical simulation and verification of LTL/CTL and quantitative temporal logic TCTL properties on reachable markings. With respect to timed analysis, ECT is not as mature as these tools. However, the compositional semantics of Reo and its extensibility make the toolset capable of analyzing timed data-aware behavior on hierarchical process models. Despite the abundance of workflow management systems and Petri net analyzing tools, at the moment none of them integrates all these facilities.

Kazhamiakin et al. [KPP06, KP06] present approaches for the modeling and analysis of data- and time-related properties of web service compositions. Here, BPEL-based service models are represented as state transition systems augmented with data and time constraints and verified using NuSMV and UPPAAL. This is suitable for the verification of service compositions already implemented, but does not support the modeling of choreographies that require data and time-aware coordination. Models and frameworks were developed to check service/process compatibility, including data-aware and time-aware business protocol compatibility in [IAE08, GPR08, GG09]. Our work generalizes these approaches by providing a graphical language that allows designers to build connectors and, thus, create valid service compositions from services that may not be directly compatible. The paper [Bea09b] provides an overview of efforts within the Sensoria project on exploiting the COWS process algebra as stepping stone towards further tooling for the design and verification of service architectures. The temporal logic UCTL dedicated to the verification of service-oriented applications, again involving COWS, and its model checker UMC, are discussed in [Bea09a]. Exogenous coordination as underlying Reo is also advocated in [BBC10], where a modal logic is proposed specific to connectors. However, at present, data and time are not supported in this approach.

With respect to the verification of Reo connectors, the work most closely to ours is the Vereofy model checker [BBKK09]. Vereofy uses two input languages, namely, Reo Scripting Language (RSL), and a guarded command language called Constraint Automata Reactive Module Language (CARML) which are textual versions of Reo and constraint automata, respectively. Scripts in these languages are automatically generated from graphical Reo models. However, Vereofy provides a format for specifying filter conditions that is less expressive, and, in contrast to our approach, does not allow join nodes and transformer channels in the circuits. Moreover, its functionality is limited to the analysis of CA, while the verification of context-sensitive models, TCA and ACA is not supported. Kemper [Kem09] presents a SAT-based approach for bounded model checking of TCA [ABBR07]. In this work, the behavior of a TCA is represented by formulae in propositional logic with linear arithmetic to be analyzed by various SAT solvers. Since TCA provide operational semantics for timed Reo, this approach can be used for model checking time properties of Reo connectors. However, to the best of our knowledge, at the moment there is no tool for generating TCA from graphical data-aware Reo circuits.

8. Conclusions

In this paper, we presented an extended approach for the mapping of Reo connectors to the process algebra mCRL2. More specifically, we provided mappings for four different semantic models of Reo and proved the correctness of the mapping for constraint automata. The semantic models of Reo that we considered here enable the analysis of different, sometimes orthogonal, aspects of connectors, such as data-awareness, context-sensitivity and blocking behavior of channels. The differences in the expressiveness of the semantic models, but also their complexity in terms of the size of the state spaces they generate, play an important role in the choice for or against one of these models. However, a formal comparison of the expressiveness of the semantic models is outside the scope of this paper.

Besides the formal mapping of Reo into the mCRL2 language, we have illustrated the practical use of our framework for dataflow analysis and evaluated the tool performance in the presence of optimization techniques. Together with other tools from ECT, our plug-in provides a user-friendly environment for graphical modeling of component/service-based systems and business processes. On the one hand, our tool releases developers from the need to encode the behavior of their systems in the specification language mCRL2 directly. On the other hand, the mCRL2 toolset supports full-featured verification for Reo. Reo has an intuitive graphical notation and workflow models designed with ECT look similar to UML activity diagrams. This will make our framework easy to use by any workflow designer.

Our future work includes tighter integration of our framework with web service technology. Thus, we plan to infer component specifications from WSDL and other similar formats. Another minor issue is related to the use of data scopes. In fact, each connector can deal with its own data type while boundary nodes on the borders of connectors can work as transformers that typecast one data type to another. Furthermore, it can be interesting to extend our tools with performance analysis using appropriate tools from the CADP toolset. We will also consider the integration of Reo with mCRL2-compatible model-based test generation tools such as JTorX[7] to automatically test channel-based service composition implementations. Finally, we plan to provide tools for Java code generation to execute coordination protocols designed with Reo. Two Java code generation prototypes for Reo are available [AKM+08]. One of them runs on a single machine and requires a precomputed CA for the whole circuit. Another one is a distributed implementation which directly implements the functionality of Reo channels and nodes, but fails to account for time details in channels and, thus, does not guarantee fairness of non-deterministic choice. For example, if an AsyncDrain channel is used in a complex synchronous circuit, this implementation will favor a request that arrives first from a remote node, while according to Reo, if requests at both source ends appear at the same time, the AsyncDrain should choose any of them non-deterministically regardless of the delay to reach its source ports. Moreover, none of the implementations deals with timed channels, data-aware channels and type checking. We believe that our work on representing various semantic models for Reo in a unified way using mCRL2 will help to overcome these problems.

Acknowledgment We are indebted to the reviewers for their feedback and constructive comment.

References

[Aal98] W.M.P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8:21–66, 1998.

[Aal99] W.M.P. van der Aalst. Woflan: a Petri-net-based workflow analyzer. *Systems Analysis Modelling Simulation*, 35:345–357, 1999.

[ABB+05] F. Arbab, C. Baier, F. de Boer, J. Rutten, and M. Sirjani. Synthesis of Reo circuits for implementation of component-connector automata specifications. In *Proc. COORDINATION 2005*, pages 236–251. LNCS 3454, 2005.

[ABBR07] F. Arbab, C. Baier, F. de Boer, and J. Rutten. Models and temporal logical specifications for timed component connectors. *Software and Systems Modeling*, 6:59–82, 2007.

[ACM+09] F. Arbab, T. Chothia, R. van der Mei, M. Sun, Y.J. Moon, and C. Verhoef. From coordination to stochastic models of QoS. In J. Field and V.T. Vasconcelos, editors, *Proc. COORDINATION 2009*, pages 268–287. LNCS 5521, 2009.

[ACSM07] F. Arbab, T. Chothia, M. Sun, and Y-J. Moon. Component connectors with QoS guarantees. In *Proc. COORDINATION 2007*, pages 286–304. LNCS 4467, 2007.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[7] http://fmt.cs.utwente.nl/redmine/wiki/jtorx/

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

[Arb96] F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Proc. COORDINATION'96*, pages 34–56. LNCS 1061, 1996.

[Arb04] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366, 2004.

[Bai05] C. Baier. Probabilistic models for Reo connector circuits. *Journal of Universal Computer Science*, 11(10):1718–1748, 2005.

[BBC10] M.A. Barbosa, L.S. Barbosa, and J.C. Campos. A coordination model for interactive components. In F. Arbab and M. Sirjani, editors, *Proc. FSEN 2009*, pages 416–430. LNCS 5961, 2010.

[BBKK09] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz. A uniform framework for modeling and verifying components and connectors. In J. Field and V.T. Vasconcelos, editors, *Proc. COORDINATION 2009*, pages 268–287. LNCS 5521, 2009.

[BBR10] J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Number 50 in Cambridge Tracts in Theoretical Computer Science. CUP, 2010.

[BCS09] M. Bonsangue, D. Clarke, and A. Silva. Automata for context-dependent connectors. In J. Field and V.T. Vasconcelos, editors, *Proc. COORDINATION 2009*, pages 184–203. LNCS 5521, 2009.

[Bea09a] M.H. ter Beek et al. CMC-UMC: A framework for the verification of abstract service oriented properties. In *Proc. SAC'09*, pages 2111–2117. ACM, 2009.

[Bea09b] L. Bocchi et al. From architectural to behavioural specification of services. *ENTCS*, 253:3–21, 2009.

[BRV04] B. Berthomieu, P.O. Ribet, and F. Vernadat. The tool TINA: Construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42, 2004.

[BSAR06] C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61:75–113, 2006.

[CCA07] D. Clarke, D. Costa, and F. Arbab. Connector coloring I: Synchronization and context dependency. *Science of Computer Programming*, 66:205–225, 2007.

[CK07] T. Chothia and J. Kleijn. Q-automata: Modelling the resource usage of concurrent components. *ENTCS*, 175:153–167, 2007.

[CKA10] B. Changizi, N. Kokash, and F. Arbab. A unified toolset for business process model formalization. In J. Happe and B. Buhnova, editors, *Proc. FESCA 2010*, pages 147–156. ENTCS, 2010.

[Cos10] D. Costa. *Formal Models for Context Dependent Connectors for Distributed Software Components and Services*. PhD thesis, Vrije Universiteit Amsterdam, 2010.

[CPLA08] D. Clarke, J. Proenca, A. Lazovik, and F. Arbab. Deconstructing Reo. In C. Canal, P. Poizat, and M. Sirjani, editors, *Proc. FOCLASA 2008*, pages 43–58. ENTCS 229, 2008.

[Gea07] J.F. Groote et al. The formal specification language `mCRL2`. In E. Brinksma et al., editor, *Methods for Modelling Software Systems*, pages 1–34. IBFI, Schloss Dagstuhl, 2007.

[GG09] N. Guermouche and C. Godart. Asynchronous timed web service-aware choreography analysis. In P. van Eck et al., editor, *Proc. CAiSE*, pages 364–378. LNCS 5565, 2009.

[GMRMS07] S. Gorton, C. Montangero, S. Reiff-Marganiec, and L. Semini. StPowla: SOA, policies and workflows. In E. Di Nitto and M. Ripeanu, editors, *Proc. ICSOC Workshops*, pages 351–362. LNCS 4907, 2007.

[GPR08] N. Guermouche, O. Perrin, and C. Ringeissen. Timed specification for web services compatibility analysis. *ENTCS*, 200:155–170, 2008.

[HCLY07] Shi-Ming Huang, Yuang-Te Chu, Shing-Han Li, and D.C. Yen. Enhancing conflict detecting mechanism for web services composition: A business process flow model transformation approach. *Science of Computer Programming*, 66(3):205–225, 2007.

[HKS+05] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J.V. den Bussche. Petri net + nested relational calculus = dataflow. In R. Meersman et al., editor, *Proc. CoopIS*, pages 220–237. LNCS 3760, 2005.

[HKS+08] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J.V. den Bussche. DFL: A dataflow language based on Petri nets and nested relational calculus. *Information Systems*, 33:261–284, 2008.

[HOP05] A. Haller, E. Oren, and S. Petkov. Survey of workflow management systems. Technical Report 2005-05-02, DERI, 2005.

[HvPS05] K. Hee van, R. Post, and L. Somers. Yet another smart process editor. In J. Feliz-Teixeira and A.E. Carvalho Brito, editors, *Proc. ESM'05*, pages 527–530, 2005.

[IAE08] M. J. Ibanez, P. Alvarez, and J. Ezpeleta. Flow and data compatibility for the correct interaction between web processes. In M. Mohammadian, editor, *Proc. CIMCA-IAWTIC-ISE*, pages 715–721. IEEE, 2008.

[JKA11] S.-S.T.Q. Jongmans, C. Krause, and F. Arbab. Encoding context-sensitivity in reo into non-context-sensitive semantic models. In *Proc. COORDINATION 2011*, pages 31–48. LNCS 6721, 2011.

[KCA10] N. Kokash, 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, pages 106–121, 2010.

[Kem09] S. Kemper. SAT-based verification for timed component connectors. *ENTCS*, 255:103–118, 2009.

[KH06] F.A. Kraemer and P. Herrmann. Service specification by composition of collaborations: An example. In *Proc. SerComp*, pages 129–133. IEEE, 2006.

[KKdV10] N. Kokash, C. Krause, and E.P. de Vink. Time and data aware analysis of graphical service models in Reo. In A. Maggiolo-Schettini, J.L. Fiadeiro, and S. Gnesi, editors, *Proc. SEFM'10*, pages 125–134. IEEE, 2010.

[KKV10a] N. Kokash, C. Krause, and E.P. de Vink. Data-aware design and verification of service composition with Reo and `mCRL2`. In *Proc. SAC 2010*, pages 2406–2413. ACM, 2010.

[KKV10b] N. Kokash, C. Krause, and E.P. de Vink. Verification of context-dependent channel-based service models. In F. de Boer, M. Bonsangue, S. Hallerstede, and M. Leuschel, editors, *Proc. FMCO 2009*, pages 21–40. LNCS 6286, 2010.

[KNP02] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In T. Field, P.G. Harrison, J.T. Bradley, and U. Harder, editors, *Proc. TOOLS 2002*, pages 200–204. LNCS 2324, 2002.

[KP06] R. Kazhamiakin and M. Pistore. Static verification of control and data in web service compositions. In *ICWS'06*, pages 83–90. IEEE, 2006.

[KPP06] R. Kazhamiakin, P. K. Pandya, and M. Pistore. Timed modelling and analysis in web service compositions. In *ARES'06*, pages 840–846. IEEE, 2006.

[Kra11] Christian Krause. *Reconfigurable Component Connectors*. PhD thesis, Leiden University, The Netherlands, 2011.

[LRST09] D. Lime, O.H. Roux, C. Seidner, and L.-M. Traonouez. Roméo: A parametric model-checker for Petri nets with stopwatches. In S. Kowalewski and A. Philippou, editors, *Proc. TACAS 2009*, pages 54–57. LNCS 5505, 2009.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MSA06] M. Mousavi, M. Sirjani, and F. Arbab. Formal semantics and analysis of component connectors in Reo. *Electronic Notes in Theoretical Computer Science*, 154(1):83–99, 2006.

[Rea03] A. Ratzer et al. CPN tools for editing, simulating, and analysing coloured Petri nets. In W.M.P. van der Aalst and E. Best, editors, *Proc. ICATPN 2003*, pages 450–462. LNCS 2679, 2003.

[RPU+07] I. Raedts, M. Petkovic, Y.S. Usenko, J.M. van der Werf, J.F. Groote, and L. Somers. Transformation of BPMN models for behaviour analysis. In J.C. Augusto, J. Barjis, and U. Ultes-Nitsche, editors, *Proc. MSVVEIS 2007*, pages 126–137. INSTICC Press, 2007.

[TAS08] N. Trčka, W. van der Aalst, and N. Sidorova. Analyzing control-flow and data-flow in workflow processes in a unified way. Technical report, Technische Universiteit Eindhoven, 2008.