Verification of Context-Dependent Channel-Based Service Models

N. Kokash[1,⋆,⋆⋆], C. Krause[1,⋆⋆⋆], and E.P. de Vink[2]

[1] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[2] Technische Universiteit Eindhoven, Den Dolech 2, Eindhoven, The Netherlands

Abstract. The paradigms of service-oriented computing and model-driven development are becoming of increasing importance in the field of software engineering. According to these paradigms, new systems are composed with added value from existing stand-alone services to support business processes across organizations. Services comprising a system but originating from various sources need to be coordinated. The Reo coordination language is a state-of-the-art computer supported approach to channel-based coordination. Reo introduces various types of channels which can be composed to build complex connectors to represent various behavioral protocols. This makes Reo suitable for the modeling of service-based business processes. In previous work we presented a framework for model checking data-aware Reo connectors using the `mCRL2` toolset. In this paper, we extend this result with a proof of correctness, evaluation of optimization techniques, and support for context-sensitive analysis.

1 Introduction

Service-oriented computing is a paradigm that is changing the way modern software is designed and developed. Services are autonomous, loosely coupled software components with publicly available interfaces that can be invoked by a client or composed by a third party to achieve a more complex goal. An important difference of service-oriented architectures compared to other architectural solutions is that the owner of a service-based system has very limited control over the services involved, as generally they run remotely by external companies which may not even know about each other. Conceptually this is similar to the idea of exogenous coordination which advocates the separation of computation (in this case, provided by services) and coordination [1].

One way to coordinate external services is to use a network of communication channels. Reo is an expressive channel-based coordination language with computer aided support. Reo introduces various types of channels which can be composed into complex connectors (also called circuits) to implement interaction protocols. Along with the graphical notation and intuitive meaning of channel

⋆ Corresponding author, email: `Natallia.Kokash@cwi.nl`
⋆⋆ Supported by IST COMPAS FP7-ICT-2007-1 project, contract number 215175
⋆⋆⋆ Supported by NWO GLANCE project WoMaLaPaDiA and SYANCO

behavior, several formal semantic models for Reo have been proposed [2,3]. 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. However, these semantic models, in particular constraint automata [2] and coloring semantics [3], require the development of special software tools to deal with them. For example, Reo animation and simulation engines [4] are developed as Eclipse plug-ins at CWI to animate and simulate the execution of Reo circuits based on coloring semantics, and Vereofy [5] is a model checking tool developed by the University of Dresden to check properties of constraint automata.

Since the tailored development of reliable verification tools is a substantial effort requiring man power over an extensive time span to mature, we chose an alternative approach. In our recent work [6], we presented a framework for specifying behavior of Reo in `mCRL2`, a specification language based on the process algebra ACP, including time and data [7]. Specifications in this language can be analyzed by an extensive set of model checking and simulation tools available in the `mCRL2` toolset. Moreover, a specification can be converted into a labelled transition system (LTS) in various formats and subsequently used as input for external model checking tools, in particular `CADP` [8]. Both `mCRL2` and `CADP` have proven their suitability for analyzing large scale industrial systems. `mCRL2` provides means to deal with algebraic data types and user-defined functions. These features are essential for enabling data-aware analysis of Reo circuits which may accept as input structured data elements from web services and may transform them (e.g., merge, duplicate, reorder) using filter and transformer channels.

We developed a conversion tool that generates `mCRL2` specifications from Reo graphical models. These specifications are generated fully automatically and do not require any manual refinement. The mapping from Reo to `mCRL2` is performed according to the constraint automata semantics of Reo. In this paper, we establish the correctness of this mapping by proving the bisimilarity of the generated `mCRL2` specification and constraint automata semantics for a given Reo circuit. Secondly, we provide experimental results that show the efficiency of the mapping that accounts for the structural information about the Reo circuit and leads to the step-wise processing of the generated specification while generating an LTS from it. Finally, we incorporate coloring information in Reo to `mCRL2` encoding. Coloring semantics have been introduced initially to provide an animated execution of Reo circuits. In contrast to constraint automata in their basic form, coloring semantics is able to express the behavior of context-dependent Reo channels, i.e., channels whose behavior depends on the states of other channels or components. A basic example of such channels is a *synchronous lossy* channel that loses data only if it cannot simultaneously dispense it. We present an extension of our conversion tool that maps Reo channels into `mCRL2` processes that explicitly propagate the information about their states to other parts of the circuit by means of typed actions.

The rest of this paper is organized as follows. In Section 2, we summarize the basics of Reo. In Section 3, we review the `mCRL2` specification language and

its toolset. In Section 4, we briefly describe the translation of Reo to mCRL2. In Section 5, we formally prove the correctness of this translation. In Section 6, we discuss two semantic models for Reo that are more expressive, namely, the coloring semantics, and translate context-aware Reo to mCRL2. In Section 7, we describe an updated conversion tool implemented as part of the Eclipse Coordination Tools (ECT) and evaluate its performance with and without optimization techniques based on the structural information about Reo circuits. In Section 8, we discuss related work. Finally, in Section 9, we give concluding remarks and outline future work.

2 The Reo coordination language

Reo is a channel-based coordination language wherein components or services are coordinated exogenously (from outside) by so-called *connectors* [9]. These connectors have a graph-like structure where the edges are user-defined communication channels and the nodes implement a standard 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. Reo allows channels to have respectively two source or sink ends. Although channels can be defined by users in Reo, a set of basic channels suffices to implement rather complex coordination protocols. One of the most basic channels in Reo is the so-called 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. The LossySync channel behaves similarly except that it always accepts data items through its source end. The data item is transferred if it can be dispensed through the sink end, and lost otherwise. The SyncDrain has two source ends and accepts data through them simultaneously. All accepted data items are lost. The AsyncDrain channel accepts data items through any of its two source ends, but never from both of them synchronously. The FIFO is an asynchronous channel with a buffer of size one. The basic set of Reo channels also includes channels that have data dependent behavior or perform data manipulation. For instance, the Filter channel loses the data item at its source end if the item does not match a certain pattern, which is defined in terms of a data constraint for a particular instance of this channel. Furthermore, data manipulation can be implemented using the Transform channel. It applies a user-defined function to the data item at its source end and yields the result at its sink end.

Channels can be joint together using nodes. A node can be of one out of three types: source, sink or mixed, depending on whether all coinciding channel ends are source ends, sink ends or a combination of both. Source and sink nodes together form the boundary of a connector, allowing interaction with its environment. Source nodes act as synchronous replicators, sink nodes as mergers. Mixed nodes combine both behaviors by atomically consuming a data item from one sink end at the time and replicating it to all source ends.

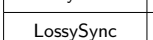
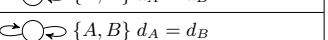
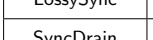
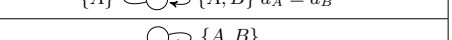
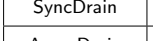
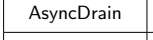
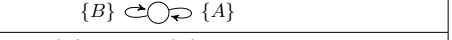
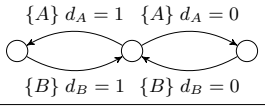
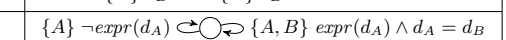
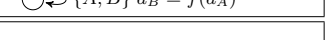
Channel name	Graphical notation	Constraint automaton
Sync		
LossySync		
SyncDrain		
AsyncDrain		
FIFO		
Filter		
Transform		
Merger		
Replicator		

Table 1. Graphical notation and semantics for channels and nodes

Semantics of Reo can be given in terms of constraint automata [2]. The transitions in constraint automata are labeled with sets of synchronously firing ports, as well as with data constraints on these ports, if desired. Figure 1 depicts the graphical notation and the constraint automata semantics for the basic channels and of the Merger and Replicator primitives, which can be used to construct nodes. Note that the constraint automaton shown for the FIFO is with respect to the data domain $Data = \{0, 1\}$. Formally, constraint automata are defined as follows.

Definition 1. *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* $s_0 \in S$*.*

For a comprehensive discussion of the constraint automata semantics of Reo we refer to [2]. The intuitive idea of constraint automata is that if the data constraint is satisfied, the corresponding transition can fire and data flow is observed at the given ports. We write $s \xrightarrow{N} t$, without constraint, for a transition indicating that while going from the state s to the state t, flow is observed at the ports in the set N.

3 The mCRL2 specification language

We provide a brief overview of the `mCRL2` specification language and toolset. For more detail we refer to [7] and to the `mCRL2` website.[3]

[3] `http://mcrl2.org/mcrl2/wiki/index.php`

The basic notion in mCRL2 is the action. Actions represent atomic events and can be parameterized 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 $|$, like the multiaction $a|b$ of simultaneous doing the action a and b. 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 $c \rightarrow p \diamond q$, where c is a data expression that evaluates to true or false;
- *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* $a@t$ indicates that multiaction a happens at time t;
- *parallel composition* or *merge* $p \parallel q$, which interleaves and synchronizes the multiactions of p with those of q;
- the *restriction operator* $\nabla_V(p)$, where V specifies which actions from p are allowed to occur, and, complementary, the *encapsulation* $\partial_H(p)$, 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 c.

The mCRL2 language provides a number of built-in data types such as boolean, natural and positive numbers, integers and real numbers. All standard arithmetic operations for them are predefined.

Custom data type definition mechanisms in mCRL2 allow users to declare new sorts, constructors and functions. A structured type in mCRL2 can be declared by a construct of the form

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

This construct defines the type S together with constructors $c_i : S_i^1 \times \ldots \times S_i^{ki} \rightarrow S$, projections $p_i^j : S \rightarrow S_i^j$, and recognizers $r_i : S \rightarrow \mathit{Bool}$. Various examples of custom type definitions can be found in the language reference section of the mCRL2 web site.

The mCRL2 toolset provides tools that allow 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 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 can be found on 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 parameterized boolean equation system (PBES) and can be solved with the appropriate tools from the toolset. Analysis at the level of LTSs is also possible by means of equivalence checking (e.g., strong and branching bisimulation or trace equivalence). In particular, the presence or absence of deadlocks/livelocks or of certain actions can be checked straightforwardly.

4 Translating Reo to mCRL2

In this section, we recall the rules for mapping Reo primitives (channels and nodes) to mCRL2 processes and briefly explain how to derive composite specifications for arbitrarilyy complex Reo connectors (cf. [6]).

Our mapping of the basic channels reflects the constraint automata semantics of Reo. The mCRL2 process corresponding to a channel, is based on two atomic actions modeling data flow on its respective ends. Analogously, we introduce a process for every node and actions for all channel ends meeting at the node. One important aspect of our encoding is that data constraints are translated faithfully. For this the actions corresponding to channel and node ends are extended with data parameters. In the context of a given connector, we assume a global datatype, which we model as the custom sort *Data* in mCRL2. In fact, most channels in Reo are agnostic to the actual type of data items they carry. Given such a global type, we can use the summation operator in mCRL2 to define data dependencies imposed by channels. The encodings for the primitives used in this paper are depicted in Table 2. Note that for the FIFO we need to define an additional datatype:

$$DataFIFO = \textbf{struct}\ empty?isEmpty \mid full(e : Data)?isFull;$$

The encoding of the FIFO channel includes a parameter of this datatype 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.

As in the constraint automata approach, we construct nodes compositionally out of the Merger and the Replicator primitive. A process for a node that behaves like an exclusive router can be defined analogously. However, in practical situations a third type of node comes in hand, which we refer to as Join here. Such a Join synchronizes all coinciding source ends and atomically forms a tuple of the data items at these ends and transfers it to the sink end. We define a binary join as

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

For handling data structures formed after passing through the Join node, we need to extend our global datatype with a notion of tuples. Since mCRL2 supports standard algebraic datatypes, this is not a problem. Throughout the rest of the

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

$$
\begin{aligned}
\mathsf{Merger} &= \Sigma_{d:Data}\ (A(d)|C(d) + B(d)|C(d)) \cdot \mathsf{Merger} \\
\mathsf{Replicator} &= \Sigma_{d:Data}\ A(d)|B(D)|C(d) \cdot \mathsf{Replicator}
\end{aligned}
$$

Table 2. mCRL2 encoding for channels and nodes

paper, we assume that the global datatype is the summation of n user-defined datatypes, which we refer to as $\mathcal{D}_1, \ldots, \mathcal{D}_n$. In concrete cases, they are inferred from the coordinated components or services. If a circuit contains one or more Join nodes, we define the global datatype as

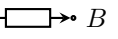
$$\mathit{Data} = \textbf{struct}\ D_1(e_1 : \mathcal{D}_1) \mid ... \mid D_n(e_n : \mathcal{D}_n) \mid \mathit{tuple}(p_1 : \mathit{Data}, p_2 : \mathit{Data})$$

This definition allows us to instantiate elements of any basic type as well as binary tuples, thus forming tree-like structures. Note, this datatype is suitable for circuits with Join nodes that have two incoming ends only. In the general case, for every Join node with k incoming ends a $\mathit{tuple}_k(p_1 : \mathit{Data}, ..., p_k : \mathit{Data})$ must be added to the definition.

After generating process definitions for all channels and nodes, we need to compose them into one joint process which models the whole connector. This is done using the following three steps:

1. Forming of the parallel composition of all channel and node processes.
2. Synchronizing of actions for coinciding channel and node ends.
3. Hiding of internal actions (optional).

Step 2 in fact involves an application of two mCRL2 operators: communication and blocking. To elucidate the composition process, we now consider a simple example. Consider the LossyFIFO connector defined as



For simplicity, we assume that the channel ends at A and B form the boundary of the connector, without explicit nodes. Thus, the system consists of two channels and one node which are represented by the following three processes:

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

For obtaining the mCRL22 process for the LossyFIFO connector, we first form the parallel composition of the aforementioned three processes, force actions corresponding to the connected channel and node ends communicate, and finally hide the actions representing the data flow at the internal node. In the mCRL2 syntax, this reads

$$\begin{aligned}\mathsf{Connector} = \rho_{N\to N\setminus\{X,Y\}}(&\\ &\partial_{\{X_1,X_2,Y_1,Y_2\}}(\\ &\quad \Gamma_{\{X_1|X_2\to X,Y_1|Y_2\to Y\}}(\\ &\quad\quad \mathsf{LossySync} \parallel \mathsf{Node} \parallel \mathsf{FIFO}\ )));\end{aligned}$$

However, this direct approach does not exploit the information about the circuit structure, and, as we show later, the further processing of the obtained specification is very inefficient. Therefore, we build up the process for a Reo connector in a stepwise fashion, i.e.,

$$\begin{aligned}\mathsf{Connector1} &= \rho_{N\to N\setminus\{Y\}}(\partial_{\{Y_1,Y_2\}}(\Gamma_{\{Y_1|Y_2\to Y\}}(\mathsf{Node} \parallel \mathsf{FIFO})));\\ \mathsf{Connector} &= \rho_{N\to N\setminus\{X\}}(\partial_{\{X_1,X_2\}}(\Gamma_{\{X_1|X_2\to X\}}(\mathsf{LossySync} \parallel \mathsf{Connector1})));\end{aligned}$$

In this version, we first compose the node and the FIFO, synchronize and hide their connected ends, and then continue with the rest of the circuit. This helps us to keep the intermediate state spaces relatively small. Note that we can use the topology of the connector to determine an order of the processes that significantly increases the runtime of the linearization algorithm. A comparison of the actual runtimes of the two approaches is discussed in Section 7.

5 Correctness of the translation

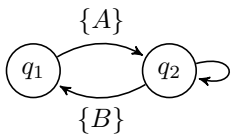
In this section, we prove the correctness of the mapping of Reo to mCRL2 discussed above. For simplicity we only consider connectors with disjoint sets of port names. Note that this does not affect the generality of our approach as we can always make these sets disjoint by renaming some of the ports.

For each state s of a given constraint automaton $\mathcal{A}$, we define the mCRL2 process $proc(\mathcal{A}, s)$ over the action set $\mathcal{P}(N)$ as

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

where $N = \prod_{x\in N} x$ represents the multiaction composed from all ports in the set. 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 $\mathcal{F}$ given by the constraint automaton and the corresponding mCRL2 processes below



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

In essence, as discussed in Section 4, the translation 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 2. *Let $\mathcal{A}_1 = (S_1, \mathcal{N}_1, \to_1, s_0^1)$, $\mathcal{A}_2 = (S_2, \mathcal{N}_2, \to_2, s_0^2)$ be two constraint automata with disjoint sets of port names $\mathcal{N}_1$ and $\mathcal{N}_2$, respectively. A port synchronization function γ is a pair of injective mappings $\gamma_1 \colon \mathcal{N} \to \mathcal{N}_1$, $\gamma_2 \colon \mathcal{N} \to \mathcal{N}_2$ from a new set of port names $\mathcal{N}$ into $\mathcal{N}_1$ and $\mathcal{N}_2$.*

In the situation above, 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 put

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

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 $\mathcal{N}$ (which is uniquely defined by injectivity of γ_1, γ_2 and disjointness of $\mathcal{N}_1$ and $\mathcal{N}_2$.

Definition 3. *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 $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2 = (S_1 \times S_2, \mathcal{N}', \to, \langle s_0^1, s_0^2 \rangle)$ where $\mathcal{N}' = \mathcal{N}_1' |_\gamma \mathcal{N}_2'$ and the transition relation $\to$ is determined by the following rules:*

$$\frac{s_1 \xrightarrow{g_1, N_1}_1 t_1 \quad N_1 \subseteq \mathcal{N}_1'}{\langle s_1, s_2 \rangle \xrightarrow{g_1, N_1} \langle t_1, s_2 \rangle} \qquad \frac{s_2 \xrightarrow{g_2, N_2}_1 t_2 \quad N_2 \subseteq \mathcal{N}_2'}{\langle s_1, s_2 \rangle \xrightarrow{g_2, N_2} \langle s_1, t_2 \rangle} \tag{3}$$

and

$$\frac{s_1 \xrightarrow{g_1, N_1}_1 t_1 \quad s_2 \xrightarrow{g_2, N_2}_1 t_2 \quad \gamma_1^{-1}(N_1) = \gamma_2^{-1}(N_2)}{\langle s_1, s_2 \rangle \xrightarrow{\gamma(g_1 \wedge g_2), N_1 |_\gamma N_2} \langle t_1, t_2 \rangle} \tag{4}$$

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 [10], has its transitions derived from the steps of its left component P_1, its right component P_2, or their synchronization:

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

The operator '|' is governed by a communication function, say γ. We have, e.g., that $(a \cdot Q) \mid (b \cdot Q')$ yields $c \cdot (Q \parallel Q')$ if $\gamma(a, b) = c$. For `mCRL2`, if not stated otherwise, the communication function is assumed to yield the inaction δ. In the context of the synchronization product $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2$ of constraint automata two aspects are important: (i) the port synchronization function γ decides which ports synchronize, hence which sets of port names are non-trivially combined, (ii) ports that are to be synchronized, i.e. $\gamma_1[\mathcal{N}] \subseteq \mathcal{N}_1$ and $\gamma_2[N] \subseteq \mathcal{N}_2$, have their names erased from the alphabet. For the first issue, we define the `mCRL2` communication function for a parallel composition as determined by a port synchronization function for the synchronized product of two constraint automata. For the second issue, we will apply a specific blocking operator determined by the port synchronization function.

Definition 4. *A port synchronization function γ with mappings $\gamma_1 \colon \mathcal{N} \to \mathcal{N}_1$, $\gamma_2 \colon \mathcal{N} \to N_2$, induces the `mCRL2` communication function γ over the alphabet $\mathcal{P}(\mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N})$ given by*

$$\gamma(N_1, N_2) = N_1 \mid_\gamma N_2 \quad \textit{if } \gamma_1^{-1}[N_1] = \gamma_2^{-1}[N_2]$$

and $\gamma(N_1, N_2) = \delta$ otherwise. The `mCRL2` process $P_1 \|_\gamma P_2$, called the γ-synchronized merge of P_1 and P_2, is given by

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

with set of blocked actions $B = \gamma_1[\mathcal{N}] \cup \gamma_2[\mathcal{N}]$, and set of synchronizations $C = \{\, \gamma_1(n) | \gamma_2(n) \mapsto n \mid n \in \mathcal{N} \,\}$ defining the communication function.

We are now in a position to formulate a soundness result for our translation with respect to the parallel composition combined with appropriate synchronization, steps 1 and 2 of our translation. Lemma 1 states 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 [11], notation $\leftrightarrows$. It is noted that from a logical point of view, branching bisimilar hence strongly bisimilar processes can be used interchangeably within the `mCRL2` toolset.

Lemma 1. *Let $\mathcal{A}_1 = (S_1, \mathcal{N}_1, \rightarrow_1, s_0^1)$ and $\mathcal{A}_2 = (S_2, \mathcal{N}_2, \rightarrow_2, s_0^2)$ be two constraint automata with disjoint sets of port names and let γ be a port synchronization function for $\mathcal{A}_1$ and $\mathcal{A}_2$. Then it holds that*

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

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

$$\mathcal{R} = \{ \, (\, proc_1(\mathcal{A}_1, s_1) \parallel_\gamma proc_2(A_2, s_2), \, proc(\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2, \langle s_1, s_2 \rangle) \,) \mid s_1 \in \mathcal{A}_1, \, s_2 \in \mathcal{A}_2 \, \}$$

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

Suppose $proc_1(s_1) \parallel_\gamma proc_2(s_2) \xrightarrow{N} P$. From the semantics of $\parallel_\gamma$ we obtain

(i) $\exists P_1$: $proc_1(s_1) \xrightarrow{N} P_1$, $N \subseteq \mathcal{N}_1'$, and $P = P_1 \parallel_\gamma proc_2(s_2)$,
(ii) $\exists P_2$: $proc_2(s_2) \xrightarrow{N} P_2$, $N \subseteq \mathcal{N}_2'$, and $P = proc_1(s_1) \parallel_\gamma P_2$, or,
(iii) $\exists P_1, P_2, N_1, N_2$: $proc_1(s_1) \xrightarrow{N_1}_1 P_1$, $proc_2(s_2) \xrightarrow{N_2}_2 P_2$, $N = N_1 \mid_\gamma N_2$ and $P = P_1 \parallel_\gamma P_2$.

By the definition of $proc_1(s_1)$ and $proc_2(s_2)$ we then have

(i) $\exists g, t_1$: $s_1 \xrightarrow{g,N}_1 t_1$, $N \subseteq \mathcal{N}_1'$, and $P = proc_1(t_1) \parallel_\gamma proc_2(s_2)$,
(ii) $\exists g, t_2$: $s_2 \xrightarrow{g,N}_2 t_2$, $N \subseteq \mathcal{N}_2'$, and $P = proc_1(s_1) \parallel_\gamma proc_2(t_2)$, or,
(iii) $\exists g_1, g_2, t_1, t_2, N_1, N_2$: $s_1 \xrightarrow{g_1,N_1}_1 t_1$, $s_2 \xrightarrow{g_2,N_2}_2 t_2$, $N = N_1 \mid_\gamma N_2$ and $P = proc_1(t_1) \parallel_\gamma proc_2(t_2)$.

Therefore, $\langle s_1, s_2 \rangle \xrightarrow{g,N} \langle t_1, s_2 \rangle$ in $\mathcal{A}_1 \parallel_\gamma \mathcal{A}_2$, thus $proc(s_1, s_2) \xrightarrow{N} proc(t_1, s_2)$, while $proc_1(t_1) \parallel_\gamma proc_2(s_2) \; \mathcal{R} \; proc(t_1, s_2)$ regarding (i). A symmetrical remark applies regarding (ii). Finally, $\langle s_1, s_2 \rangle \xrightarrow{g,N} \langle t_1, t_2 \rangle$ with $g = \gamma(g_1 \wedge g_2)$, $proc(s_1, s_2) \xrightarrow{N} proc(t_1, t_2)$, while $proc_1(t_1) \parallel_\gamma proc_2(t_2) \; \mathcal{R} \; proc(t_1, t_2)$ regarding (iii).

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

(i) $N \subseteq \mathcal{N}_1'$ and $\exists g, t_1$: $\langle s_1, s_2 \rangle \xrightarrow{g,N} \langle t_1, s_2 \rangle$ based on $s_1 \xrightarrow{g,N}_1 t_1$, and $P = proc(t_1, s_2)$,
(ii) $N \subseteq \mathcal{N}_2'$ and $\exists g, t_2$: $\langle s_1, s_2 \rangle \xrightarrow{g,N} \langle s_1, t_2 \rangle$ based on $s_2 \xrightarrow{g,N}_2 t_2$, and $P = proc(s_1, t_2)$, or,
(iii) $N \not\subseteq \mathcal{N}_1$, $N \not\subseteq \mathcal{N}_2$ and $\exists g, g_1, g_2, t_1, t_2, N_1, N_2$: $\langle s_1, s_2 \rangle \xrightarrow{g,N} \langle t_1, t_2 \rangle$ based on $s_1 \xrightarrow{g_1,N_1}_1 t_1$ and $s_2 \xrightarrow{g_2,N_2}_2 t_2$ with $N = N_1 \mid_\gamma N_2$, and $P = proc(t_1, s_2)$.

Then, we have $proc_1(s_1) \parallel_\gamma proc_2(s_2) \xrightarrow{N} proc_1(t_1) \parallel_\gamma proc_2(s_2)$ and $proc_1(t_1) \parallel_\gamma proc_2(s_2) \; \mathcal{R} \; proc(t_1, s_2)$ in case of (i), a symmetrical observation in case of (ii), and $proc_1(s_1) \parallel_\gamma proc_2(s_2) \xrightarrow{N} proc_1(t_1) \parallel_\gamma proc_2(t_2)$ and $proc_1(t_1) \parallel_\gamma proc_2(t_2) \; \mathcal{R} \; proc_{12}(t_1, t_2)$ 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 $proc(\mathcal{A}_1, s_1) \parallel_\gamma proc(\mathcal{A}_2, s_2) \leftrightarrows proc(\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2, \langle s_1, s_2 \rangle$. □

Note that the above result, in the present setting of a parallel construct involving a port synchronization function, shows strong bisimilarity, whereas the original result of [2] claims, for general composition, language equivalence with respect to timed data streams.

Next, we turn the final step of the translation, the optional hiding of internal flow corresponding to port names of mixed nodes. In the constraint automaton representation this amounts to restricting the observable flow, in `mCRL2` it can be captured by a proper renaming function.

Definition 5. *Let $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0)$ be a constraint automaton and $C \subseteq \mathcal{N}$ a subset of ports. The C-restricted constraint automaton $\mathcal{A}\backslash C$ is given by $\mathcal{A}\backslash C = (S, \mathcal{N}\backslash C, \rightarrow_C, s_0)$ where $\rightarrow_C$ is given by*

$$s \xrightarrow{g,N}_C t \quad \textit{iff} \quad \exists f\, \exists M \subseteq \mathcal{N} \colon s \xrightarrow{f,M} t \wedge g = \exists C.f \wedge N = M\backslash C$$

Here, the constraint $\exists C.f$ expresses existential quantification of the port names in C for the constraint f, cf. [2]. We have that the C-restricted automaton $\mathcal{A}\backslash C$ has the same transitions as the automaton $\mathcal{A}$, but it hides the port names from C as these are considered to be internal for the underlying Reo connector. Therefore, the corresponding renaming for `mCRL2` processes needs to delete from each set of port names those in C. We have the following correctness result.

Lemma 2. *Let $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0)$ be a constraint automaton and $C \subseteq \mathcal{N}$ a subset of ports. Then it holds that*

$$proc(\mathcal{A}\backslash C, s) \leftrightarrows \varrho_C(proc(\mathcal{A}, s))$$

where $\varrho_C \colon \mathcal{N} \rightarrow \mathcal{N}$ is the renaming $N \mapsto N\backslash C$ for $N \in \mathcal{N}$.

Proof. It can be checked, similar as for Lemma 1, that the relation

$$\mathcal{R} = \{\, (proc(\mathcal{A}\backslash C, s),\ \varrho_C(proc(\mathcal{A}, s))) \mid s \in \mathcal{A} \,\}$$

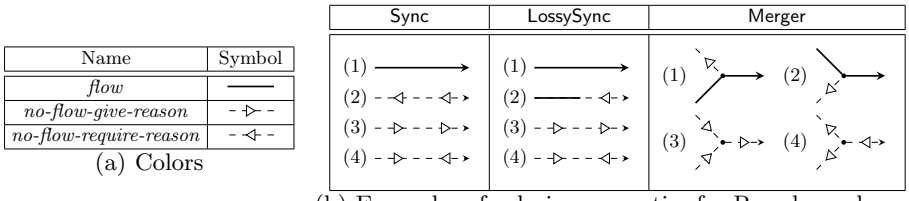
is a strong bisimulation relation by chasing the transfer properties. □

The hiding operator as introduced in [2] differs from the hiding operator presented here. In [2], in a context of language equivalence for timed data streams, an arbitrary number of transitions with flow exclusively over hidden ports are combined with a single observable transition. In our set-up, a computation of the C-restricted automaton corresponds transition-by-transition to a computation of the unrestricted automaton. Note that the minimization of the C-restricted automaton by aggregating several transitions into a weak one can be done afterwards. This operation still preserves branching bisimulation.

6 Coloring semantics

The constraint automata semantics used in the previous section has a major drawback: it cannot model context-dependency. For example, the LossySync

channel is not correctly represented as its constraint automaton can pass or lose data non-deterministically, whereas according to its informal semantics the passing of data has priority over losing. To cope with this problem, different semantical models have been introduced. One of them is the so-called *coloring* semantics. The basic idea in this model is to associate flow and no-flow colors to channel ends. Clarke et al. showed in [3] that one *flow* color and two *no-flow* colors are sufficient to model context-dependency such as required by the LossySync. The names and graphical representations of these colors are shown in Figure 1(a).



(a) Colors

(b) Examples of coloring semantics for Reo channels and nodes

Fig. 1. Colors and examples of coloring semantics for Reo channels and nodes

Valid behaviors of channels are then expressed as colorings of their respective ends. Figure 1(b) depicts 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 node 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.[4]

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.

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 depicts 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.

To deal with context-dependency in our encoding in `mCRL2` we incorporate the coloring model. We encode the different colors as simple data parameters of

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

actions. We therefore introduce a new datatype

$$\mathit{Colored} \;=\; \mathbf{struct}\ \mathit{flow}(\mathit{data} : \mathit{Data}) \mid \mathit{noflowG} \mid \mathit{noflowR},$$

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

$$\begin{aligned}
\mathsf{Sync} = \big(&\Sigma_{d:\mathit{Data}}\ A(\mathit{flow}(d)) \mid B(\mathit{flow}(d)) \;+ && (1)\\
&A(\mathit{noflowR}) \mid B(\mathit{noflowG}) \;+ && (2)\\
&A(\mathit{noflowG}) \mid B(\mathit{noflowR}) \;+ && (3)\\
&A(\mathit{noflowG}) \mid B(\mathit{noflowG})\big) \cdot \mathsf{Sync}; && (4)
\end{aligned}$$

where each line corresponds to a coloring in Fig 1(b). In the same way, the LossySync can be specified as

$$\begin{aligned}
\mathsf{LossySync} = \big(&\Sigma_{d:\mathit{Data}}\ A(\mathit{flow}(d)) \mid B(\mathit{flow}(d)) \;+ && (1)\\
&A(\mathit{flow}) \mid B(\mathit{noflowG}) \;+ && (2)\\
&A(\mathit{noflowG}) \mid B(\mathit{noflowR}) \;+ && (3)\\
&A(\mathit{noflowG}) \mid B(\mathit{noflowG})\big) \cdot \mathsf{LossySync}; && (4)
\end{aligned}$$

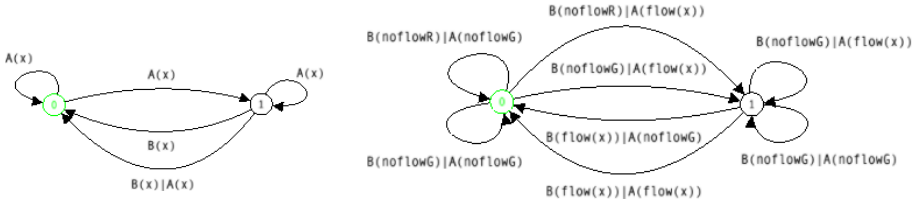
and finally, the Merger can be encoded as

$$\begin{aligned}
\mathsf{Merger} = \big(&\Sigma_{d:\mathit{Data}}\ A(\mathit{flow}(d)) \mid B(\mathit{noflowG}) \mid C(\mathit{flow}(d)) \;+ && (1)\\
&\Sigma_{d:\mathit{Data}}\ A(\mathit{noflowG}) \mid B(\mathit{flow}(d)) \mid C(\mathit{flow}(d)) \;+ && (2)\\
&A(\mathit{noflowR}) \mid B(\mathit{noflowR}) \mid C(\mathit{noflowG}) \;+ && (3)\\
&A(\mathit{noflowG}) \mid B(\mathit{noflowG}) \mid C(\mathit{noflowR})\big) \cdot \mathsf{Merger}; && (4)
\end{aligned}$$

The other channels are encoded analogously.

The LossyFIFO connector is a classical example where context-dependency is required (cf. [3]). Fig. 2 depicts the corresponding labeled transition systems for the basic constraint automata encoding (a), as well as the encoding based on the coloring semantics (b). For simplicity, we use the singleton set $\mathit{Data} = \{x\}$ as data domain. The crucial point here is that in the initial state 0, the constraint automata version can lose data (loop $A(x)$), which is an unintended behavior. However, in the coloring encoding, there is no such behavior.

Using the coloring model we can properly represent context-dependency in `mCRL2`. In contrast to [3], 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.



(a) CA encoding (b) Coloring encoding

Fig. 2. LTS of the LossyFIFO connector (screenshots from `ltsgraph`)

7 Implementation

We implemented the conversion from Reo to `mCRL2` discussed above as an extension to the Eclipse Coordination Tools (ECT), see [4]. 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, 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 [5] or PRISM [12].

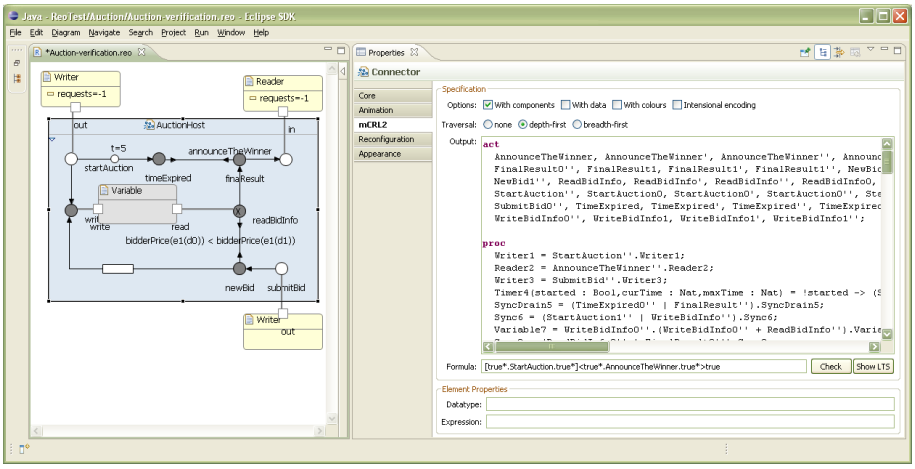


Fig. 3. Reo model of an auction process and its `mCRL2` specification

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

From our conversion tool, an mCRL2 specification can be obtained automatically from any Reo circuit simply by selecting it in the graphical Reo editor. A screenshot of the tool is shown in Figure 3. The code generation can be customized using various options. For instance, enabling the option *with components* will add and incorporate process definitions for the components attached at the boundary of a connector. The option *with data* enables the data-aware encoding. If not enabled, data parameters and constraints are omitted. Furthermore, the option *with colors* can be used to add support for context-dependency as described in Section 6. Moreover, 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 using the same interface. Note that they are saved as annotations in the Reo model and are automatically merged in when generating the final mCRL2 specification. This way we can ensure that the mCRL2 code can be regenerated at any point without manual changes if desired.

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 from mCRL2 code, `lps2lts` and `lpsconvert` for generating and minimizing labeled transitions systems, `lps2pbes` for model checking formulas specified in modal μ-calculus, and finally `ltsgraph` for visualizing state spaces. Related verification tools, such as CADP[6] can be integrated in a similar fashion since they share the same format for labeled transitions systems. The integration with CADP in particular belongs to our future work.

In our encoding of Reo in mCRL2 we translate every primitive (channel, node or component) to a separate process, 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 between all these processes is rather local, e.g. a channel communicates only with its source and target nodes.

Our experiments show that the direct approach of naively running all processes in parallel and then performing the communication, synchronization and optionally the hiding operator leads to a state space explosion during the linearization. To overcome this problem, we add processes one by one and immediately apply the aforementioned mCRL2 operators. We use the topology information of the connector to determine what processes communicate directly with each other. This leads to a much faster linearization process. In particular, we found out that a traversal over the connector graph is well-suited for this problem. In our experiments, an optimized depth-first traversal showed the best results.

As an example we tested the so-called Counter circuit which consists of an exclusive router with n outputs, each of the connected to another FIFO, which in turn are synchronized using a SyncDrain at their sink ends. Here we are interested only in two actions: data arriving at the source end of the exclusive router and the synchronized firing of the SyncDrains. The resulting transition system consists of

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

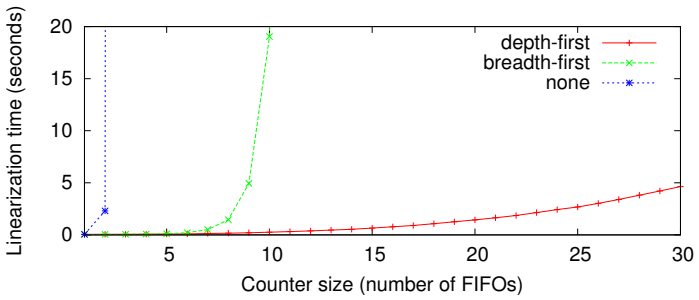


Fig. 4. Benchmarks for different encodings of the Counter circuit

n states and n transitions. Benchmarks[7] of the different optimzations is depicted in Figure 4. The linearization using depth-first traversal took less than 5 seconds for the counter with 30 FIFOs. The breadth-first approach was still able to handle 10 FIFOs in 20 seconds. However, without any optimizations `mCRL2` needed more than 20 minutes to process the counter with just 3 FIFOs.

8 Related Work

In this section, we compare our framework to other tools for analyzing Reo connectors. For an overview of related work with respect to the application of our tool to business process and web service composition analysis refer to [6].

The tool most closely related to the plug-in presented in this paper is Vereofy [5], a model checking tool developed at the University of Dresden for the analysis of Reo connectors. Vereofy uses two input languages, the 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 and are used for the verification of circuit properties expressed in LTL and CTL-like logics. The main advantage of the tool comparing to our work is that it can generate clear counterexamples and show them as paths on the initial Reo circuit, while the counterexamples in `mCRL2` may be huge and not very useful. However, in contrast to our approach, Vereofy does not support context-dependent and transformer channels, provides a format for specifying filter conditions that is less expressive than ours, and does not allow join nodes in the circuits. Moreover, it expects the user to define a global data domain eligible to all connectors and components in the model instead of generating it automatically, and cannot handle recursive type definitions as e.g., we need for dealing with join nodes.

[7] Benchmarks were taken on a standard machine with 4 cores and 8GB memory, running Linux 2.6.27 and the development version of `mCRL2` (revision 7467).

Khosravi et al. [13] establishes a mapping of Reo to Alloy, a lightweight modeling language based on first-order relational logic. To check the correctness of a circuit, the desired properties are expressed in terms of assertions which are closely related to LTL and checked by the Alloy Analyzer. The approach deals with context dependency in Reo by defining special relations that enforce maximal progress property in circuit execution. However, the actual values of data passed through the channels are not considered in this work. Moreover, the authors admit to have considerable problems with performance.

Bonsangue and Izadi [14] defined an operational semantics of context-dependent Reo connectors in terms of Büchi automata and generalized standard automata based model checking algorithms to enable verification of LTL formulas for Reo connectors. However, this work is purely theoretical is not supported by any existing software tool.

Kemper [15] presented a SAT-based approach for bounded model checking of timed constraint automata (TCA) [16]. In this work, the behavior of TCA is represented as formula in propositional logic with linear arithmetic which can 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, at the moment there is no tool for generating TCA from graphical Reo circuits. The development of such a plug-in for data-aware Reo will require tools for analyzing data constraints and functions used in filter and transformer channels. In our work, we map each channel to a process in the process algebra `mCRL2` separately, and exploit the functionality provided by the `mCRL2` toolset to obtain a semantic model of the whole circuit in terms of LTS where transitions are labeled with names parameterized with data observed in these ports. Since the `mCRL2` toolset supports time analysis, the extension of our conversion tool with the ability to deal with timer channels is straightforward and belongs to our future work.

9 Conclusions

In this paper, we presented an extended approach for mapping Reo connectors to process algebra `mCRL2`. More specifically, we proved the correctness of the mapping, extended the conversion tool with the ability to deal with context-sensitive Reo, and evaluated the tool performance in the presence of optimization techniques.

The `mCRL2` toolset supports efficient full-featured model checking for Reo. 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, which releases developers from the need to encode the behavior of their systems in the specification language of `mCRL2` directly.

As our future work, we plan to extend our approach by incorporating timer channels [16] according to their semantics in terms of TCA. This will enable the analysis of timed properties for channel-based service models. Another direction of our research is an extension of Reo semantics with various actions observable

on channel ports. In particular, this will allow us to model data flow within synchronous regions of a connector and, given time delays for each channel, estimate total delays of the circuits.

References

1. Arbab, F.: The IWIM model for coordination of concurrent activities. In Ciancarini, P., Hankin, C., eds.: Proc. COORDINATION'96, LNCS 1061 (1996) 34–56
2. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Science of Computer Programming **61** (2006) 75–113
3. Clarke, D., Costa, D., Arbab, F.: Connector coloring I: Synchronization and context dependency. Science of Computer Programming **66**(3) (2007) 205–225
4. Arbab, F., Koehler, C., Maraikar, Z., Moon, Y., Proenca, J.: Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. Tool demo session at FACS '08 (2008)
5. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A uniform framework for modeling and verifying components and connectors. In Field, J., Thudichum Vasconcelos, V., eds.: Proc. COORDINATION 2009, LNCS 5521 (2009) 268–287
6. Kokash, N., Krause, C., de Vink, E.: Data-aware design and verification of service composition with Reo and mCRL2. In: Proc. SAC 2010, Sierre, March 21–26, 2010, ACM (2010) To appear.
7. Groote, J., Mathijssen, A., Reniers, M., Usenko, Y., van Weerdenburg, M.: The formal specification language `mCRL2`. In Brinksma, E., Harel, D., Mader, A., Stevens, P., Wieringa, R., eds.: Methods for Modelling Software Systems. IBFI, Schloss Dagstuhl (2007)
8. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In Damm, W., Hermanns, H., eds.: Proc. CAV 2007, LNCS 4590 (2007) 158–163
9. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science **14** (2004) 329–366
10. Baeten, J., Basten, T., Reniers, M.: Process Algebra: Equational Theories of Communicating Processes. Number 50 in Cambridge Tracts in Theoretical Computer Science. CUP (2010)
11. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
12. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic Symbolic Model Checker. In: Proc. TOOLS'02, LNCS 2324 (2002) 200–204
13. Khosravi, R., Sirjani, M., Asoudeh, N., Sahebi, S., Iravanchi, H.: Modeling and analysis of Reo connectors using Alloy. In Lea, D., Zavattaro, G., eds.: Proc. COORDINATION 2008, LNCS 5052 (2008) 169–183
14. Bonsangue, M., Izadi, M.: Automata based model checking for reo connectors. In: Proc. FSEN'09, LNCS 5961 (2010) 260–275
15. Kemper, S.: Sat-based verification for timed component connectors. Electronic Notes in Theoretical Computer Science (ENTCS) **255** (2009) 103–118
16. Arbab, F., Baier, C., de Boer, F., Rutten, J.: Models and temporal logical specifications for timed component connectors. Software and Systems Modeling **6**(1) (2007) 59–82