ReoLive: Analysing Connectors in your Browser

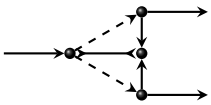
Rúben Cruz and José Proença

HASLab/INESC TEC, Universidade do Minho, Portugal
rubenamcruz@gmail.com jose.proenca@di.uminho.pt

Abstract. Connectors describe how to combine independent components by restricting the possible interactions between their interfaces. In this work, connectors are specified using an existing calculus of connectors for Reo connectors. Currently there are no tools to automatically analyse these connectors, other than a type-checker for a embedded domain specific language. A collection of tools for different variations of Reo connectors exists, but most use a heavy Eclipse-based framework that is not actively supported.
We propose a set of web-based tools for analysing connectors—named ReoLive—requiring only an offline Internet browser with JavaScript support, which also supports a client-server architecture for more complex operations. We also show that the analysis included in ReoLive are correct, by formalising the encoding of the connector calculus into Port Automata and into mCRL2 programs. We include extensions that generate such automata, mCRL2 processes, and graphical representations of instances of connectors, developed in the Scala language and compiled into JavaScript. The resulting framework is publicly available, and can be easily experimented without any installation or a running server.

1 Introduction

Proença and Clarke [9] investigated how one can specify and combine connector families, and how to check if the interfaces of these families match. Their core calculus is a monoidal category, where connectors are morphisms composed sequentially with the morphism composition ';', and in parallel with the tensor operator '$\oplus$'. This calculus was formalized with a tile semantics that describes the behaviour of a connector, and how to combine tiles between two connectors.
We pursue this work by building tools to analyse and verify a calculus of Reo connectors, focusing on its subset without variability. More concretely, we build a framework—ReoLive—that draws instances of connectors, and encodes connectors into automata and into a process algebra used by the mCRL2 model checker. This paper formally shows the correctness of these encodings, closely following the encoding of Reo connectors (seen as Constraint Automata) into mCRL2 by Kokash et al. [7].
Consider the connector in Fig. 1, known in the literature as the exclusive router. The left side presents its usual graphical representation, while the right side uses its representation in the calculus of connectors used in this paper, c.f. [9]. Intuitively, each basic element of the calculus is



Δ; $\Delta \oplus \mathsf{id}$;
$(\mathsf{lossy}; \Delta) \oplus (\mathsf{lossy}; \Delta) \oplus \mathsf{id}$;
$\mathsf{id} \oplus \nabla \oplus \mathsf{id} \oplus \mathsf{id}$;
$\mathsf{id} \oplus \mathsf{id} \oplus \gamma_{1,1}$;
$\mathsf{id} \oplus \mathsf{drain} \oplus \mathsf{id}$

Fig. 1: The exclusive router connector: its graphical representation (left) and its algebraic representation using the calculus of Reo connectors (right).

a primitive connector with a fixed sequence of source and sink ports, composed sequentially with ';' and composed in parallel with $\oplus$. More details on this calculus will be given in the next section.
The key challenges of this paper consist of presenting a framework to analyse connectors specified in this calculus, providing a set of different *widgets* that help the developer understand the graphical structure and its semantics. More specifically, the ReoLive framework receives algebraic specifications of connectors and (1) calculates and depicts a graphical representation with a easy-to-understand layout, (2) calculates and depicts an automata representing its semantics, based on constraint automata [2] without data constraints, and (3) produces a mCRL2 specification [5] that can be used for model checking with external tools. While the first contribution is less scientific and mainly technical, the other two contributions include correctness proofs, based on the formalisation of the encodings into automata and mCRL2.

Section 2 formalises Reo connectors using this calculus of connector. Section 3 translates the calculus into port automata, and Section 4 into mCRL2, following the work by Kokash et al. [7]. Section 5 describes the ReoLive framework for our calculus of connector, and Section 6 concludes and discusses future work.

2 Calculus of Reo Connector (CRC)

The input of our ReoLive framework are Reo connectors [1] specified using a calculus of connectors, following Proença and Clarke [9]. We start by describing this calculus, disregarding the notion of families presented originally, and will later show that they are indeed equivalent to two other existing semantic models: Port Automata (Section 3) and mCRL2 programs (Section 4).

2.1 Syntax

The syntax of a core connector is given by the grammar in Fig. 2. We use a simplified version from our previous publication [9] by using natural numbers for the input and output interfaces, where the tensor is the sum. This makes the category of our connectors more specific—a Prop category with traces [8]. This simplification has been made also in our

$$
\begin{array}{lll}
c ::= \mathsf{id}_n & \text{identities} & \\
\mid \gamma_{n,m} & \text{symmetries} & \\
\mid p \in \mathcal{P} & \text{primitive connectors} & \\
\mid c_1 \,;\, c_2 & \text{sequential composition} & \\
\mid c_1 \oplus c_2 & \text{parallel composition} & \\
\mid \mathsf{Tr}_n(c) & \text{traces (feedback loops)} &
\end{array}
\qquad
\begin{array}{lll}
p \in \mathcal{P} ::= \Delta_n & \text{duplicator into } n \text{ ports} & \\
\mid \nabla_n & \text{merger of } n \text{ inputs} & \\
\mid \mathsf{drain} & \text{synchronous drain} & \\
\mid \mathsf{fifo} & \text{buffer} & \\
\mid \ldots & \text{user-defined connectors} &
\end{array}
$$

Fig. 2: Grammar for core connectors, where $n, m \in \mathbb{N}$.

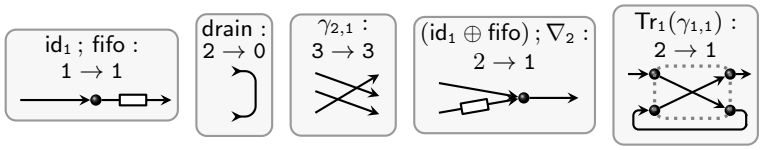


Fig. 3: Connectors, their interfaces, and their visualisation.

previous work, when describing the implementation of a type-inference algorithm.

Fig. 3 depicts some examples of connectors. Each box contains (1) a connector on top, (2) its interfaces in the middle, and (3) its visual representation below depicting inputs on the left and outputs on the right. Intuitively, each connector has a sequence of input ports and a sequence of output ports, which we number incrementally from 1. Composing two connectors sequentially $c_1; c_2$ means connecting the i-th sink port c_1 to the i-th source port of c_2, for every sink port of c_1 and source port of c_2; composing connectors in parallel $c_1 \oplus c_2$ means combining all source and sink ports of both c_1 and c_2; wrapping a connector c by a trace over n means connecting the last n sink ports of c to its last n source ports. The semantics of Reo connectors, written using this calculus, uses the Tile Model [4], following the original publication of this calculus [9].

The sintax and semantics of the calculus of connector families is not introduced in this document, as it is not refered throughout the document (except in Section 5). In [9] we can find a more detailed description of this calculus.

2.2 Tile Semantics

Each connector in the Tile Model consists of a set of tiles, one for each possible behaviour, as defined in Fig. 4. Each of these tiles contains at most 4 morphisms between shared objects, belonging to two different categories over the same objects: natural numbers, which we call $\mathcal{H}$ for the a *horizontal category* and $\mathcal{V}$ for a *vertical category*. The horizontal category $\mathcal{H}$ is the category of connectors used for the our connector calculus—with a tensor, symmetries, and traces. The vertical category $\mathcal{V}$ is a new category with the same objects $\mathbb{N}$, and with only the morphisms $\mathsf{fl} : 1 \to 1$ and $\mathsf{nofl} : 1 \to 1$, also with a tensor product, where nofl acts as the identity and the composition is represented by '$\circ$'.

$$\mathsf{id}_1 = \left\{ \mathsf{id}_1 \xrightarrow[\mathsf{fl}]{\mathsf{fl}} \mathsf{id}_1 \,,\; \mathsf{id}_1 \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{id}_1 \right\}$$

$$\gamma_{1,1} = \left\{ \gamma_{1,1} \xrightarrow[\mathsf{nofl}\oplus\mathsf{fl}]{\mathsf{fl}\oplus\mathsf{nofl}} \gamma_{1,1} \,,\; \gamma_{1,1} \xrightarrow[\mathsf{fl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{fl}} \gamma_{1,1} \,,\; \gamma_{1,1} \xrightarrow[\mathsf{fl}\oplus\mathsf{fl}]{\mathsf{fl}\oplus\mathsf{fl}} \gamma_{1,1} \,,\; \gamma_{1,1} \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{nofl}} \gamma_{1,1} \right\}$$

$$\Delta_2 = \left\{ \Delta_2 \xrightarrow[\mathsf{fl}\oplus\mathsf{fl}]{\mathsf{fl}} \Delta_2 \,,\; \Delta_2 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}} \Delta_2 \right\}$$

$$\nabla_2 = \left\{ \nabla_2 \xrightarrow[\mathsf{fl}]{\mathsf{fl}\oplus\mathsf{nofl}} \nabla_2 \,,\; \nabla_2 \xrightarrow[\mathsf{fl}]{\mathsf{nofl}\oplus\mathsf{fl}} \nabla_2 \,,\; \nabla_2 \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{nofl}} \nabla_2 \right\}$$

$$\mathsf{drain} = \left\{ \mathsf{drain} \xrightarrow[\mathsf{fl}]{\mathsf{fl}} \mathsf{drain} \,,\; \mathsf{drain} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{drain} \right\}$$

$$\mathsf{lossy} = \left\{ \mathsf{lossy} \xrightarrow[\mathsf{fl}]{\mathsf{fl}} \mathsf{lossy} \,,\; \mathsf{lossy} \xrightarrow[\mathsf{nofl}]{\mathsf{fl}} \mathsf{lossy} \,,\; \mathsf{lossy} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{lossy} \right\}$$

$$\mathsf{fifo} = \left\{ \mathsf{fifo} \xrightarrow[\mathsf{nofl}]{\mathsf{fl}} \mathsf{fifofull} \,,\; \mathsf{fifo} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{fifo} \right\}$$

$$\mathsf{fifofull} = \left\{ \mathsf{fifofull} \xrightarrow[\mathsf{fl}]{\mathsf{nofl}} \mathsf{fifo} \,,\; \mathsf{fifofull} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{fifofull} \right\}$$

Fig. 4: Behaviour of primitive connectors using tiles.

Composing Tiles Tiles can be composed in three ways: in parallel with '$\oplus$', horizontally with ';', and vertically with '$\circ$'.

$$c_1 \xrightarrow[v]{v_1} c_2 \,;\, c_1' \xrightarrow[v_2]{v} c_2' \;=\; (c_1; c_1') \xrightarrow[v_2]{v_1} (c_2; c_2') \qquad \text{(horizontal)}$$

$$c_1 \xrightarrow[v_2]{v_1} c \circ c \xrightarrow[v_2']{v_1'} c_2 \;=\; c_1 \xrightarrow[v_2' \circ v_2]{v_1' \circ v_1} c_2 \qquad \text{(vertical)}$$

$$c_1 \xrightarrow[v_2]{v_1} c_2 \oplus c_1' \xrightarrow[v_2']{v_1'} c_2' \;=\; c_1 \oplus c_2 \xrightarrow[v_2 \oplus v_2']{v_1 \oplus v_1'} c_1' \oplus c_2' \qquad \text{(parallel)}$$

For example, the tiles $t_l = \mathsf{lossy} \xrightarrow[\mathsf{fl}]{\mathsf{fl}} \mathsf{lossy}$ and $t_f = \mathsf{fifo} \xrightarrow[\mathsf{nofl}]{\mathsf{fl}} \mathsf{fifofull}$ can be composed horizontally producing the new tile $t_l; t_f = (\mathsf{lossy}; \mathsf{fifo}) \xrightarrow[\mathsf{nofl}]{\mathsf{fl}} (\mathsf{lossy}; \mathsf{fifofull})$. This new tile captures data going through the lossy and into the fifo. Similarly, t_l can be composed vertically with the tile $t_l' = \mathsf{lossy} \xrightarrow[\mathsf{nofl}]{\mathsf{fl}} \mathsf{lossy}$ yielding the new tile $t_l \circ t_l' = \mathsf{lossy} \xrightarrow[\mathsf{fl}\circ\mathsf{nofl}]{\mathsf{fl}\circ\mathsf{fl}} \mathsf{lossy}$, which captures two steps of the same lossy: first by having data flowing from its source to its sink, and later by having dataflow only on its source end.

3 Connectors as Port Automata

The semantics of the calculus of Reo connectors (CRC) is given by a set of tiles. This section encodes the tile semantics of CRC as Port Automata [6], which can be regarded as data-agnostic Constraint Automata [2], showing this encoding is correct.

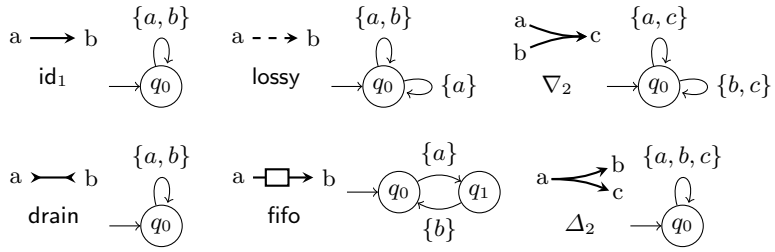


Fig. 5: Port Automata of primitive connectors.

3.1 Port Automata (PA)

Following Koehler and Clarke [6], composing two automata is done by the *product* operation $\bowtie$, forcing shared ports to occur together, while hiding ports from a connector removes them from the transitions, disallowing further communications. We define *port substitution* of a by b in an automaton $\mathcal{A}$ as the automaton $\mathcal{A}\{a \mapsto b\} = (Q, N\{a \mapsto b\}, \dashrightarrow, q_0)$, where $q_i \overset{X\{a\mapsto b\}}{\dashrightarrow} q_j$ iff $q_i \xrightarrow{X} q_j$, and $X\{a \mapsto b\}$ denotes the set X replacing a by b.

For simplicity, we write $q_i \xrightarrow{N} q_j$ to denote $\rightarrow (q_i, N, q_j)$. Fig. 5 depicts examples of a set of primitive automata commonly found in the literature, including also the corresponding notation in our calculus.

3.2 Encoding CRC into Port Automata

The semantics of CRC is given by the Tiles Model, where a tile $c_1 \xrightarrow[snk]{src} c_2$ means that the connector c_1 can evolve to a new state given by the connector c_2, by firing its source ports based on *src* and its sink ports based on *snk*. Here *src* and *snk* are morphisms built by composing simpler morphisms fl and nofl, indicating which ports have flow and no-flow.

The encoding of a connector c into a PA is written as $\mathcal{PA}(c)$, defined below. Each port is a pair (n, s) where $n \in \mathbb{N}$ is the order number of its source or sink node, and $s \in \{\mathsf{sr}, \mathsf{sk}, \mathsf{mx}\}$ is a constant that marks it as being a source (sr) or a sink (sk) port, or temporarily marking it as a mixed port during composition.

Definition 1 (Tiles of a connector). *Given a core connector c, we write $T(c)$ to represent all tiles for c and for the reachable states from c. Formally, $T(c)$ is the smallest set such that, for every tile $t = \left(c \xrightarrow[sk]{sr} c'\right)$ we have that $t \in T(c)$ and $T(c') \subseteq T(c)$.*

Definition 2 (Reachable connectors). *Given a connector c, we write $Reach(c)$ to represent all reachable connectors from c, i.e., $Reach(c)$ is the smallest set such that $c \in Reach(c)$, and for every tile $c \xrightarrow[sk]{sr} c'$ we have that $Reach(c') \subseteq Reach(c)$.*

Definition 3 (Encoding $\mathcal{PA}(c)$). *Let c be a connector from n to m. Its port automaton $\mathcal{PA}(c)$ is $(Q, N, \rightarrow, q_0)$ where*

- $Q = Reach(c)$
- $N = \{(i, \mathsf{sr}) \mid i \in \{1 \dots n\}\} \cup \{(j, \mathsf{sk}) \mid j \in \{1 \dots m\}\}$
- $q \xrightarrow{X_{\mathsf{sr}} \cup X_{\mathsf{sk}}} q' \Leftrightarrow \exists t \in T(c) : t = c_1 \xrightarrow[snk]{src} c_2 \ \wedge$
  $X_{\mathsf{sr}} = \{(i, \mathsf{sr}) \mid src = v_1 \oplus \dots \oplus v_n, i \in \{1 \dots n\}, v_i = \mathit{fl}\}$
  $X_{\mathsf{sk}} = \{(i, \mathsf{sk}) \mid snk = v_1 \oplus \dots \oplus v_m, j \in \{1 \dots m\}, v_j = \mathit{fl}\}$

For example, the fifo channel can be encoded as $\mathcal{PA}(\mathsf{fifo}) = (\{\mathsf{fifo}, \mathsf{fifofull}\}, \{(1, \mathsf{sr}), (1, \mathsf{sk})\}, \rightarrow, \mathsf{fifo})$, where

$$\mathsf{fifo} \xrightarrow{(1,\mathsf{sr})} \mathsf{fifofull} \quad \mathsf{fifofull} \xrightarrow{(1,\mathsf{sk})} \mathsf{fifo} \quad \mathsf{fifo} \xrightarrow{\emptyset} \mathsf{fifo} \quad \mathsf{fifofull} \xrightarrow{\emptyset} \mathsf{fifofull}.$$

3.3 Correctness of $\mathcal{PA}(\cdot)$

We defined how to encode any connector c into a PA $\mathcal{PA}(c)$. We say this encoding is correct with respect to an automaton A if $\mathcal{PA}(c)$ is **strongly bisimilar** to A, written $\mathcal{PA}(c) \approx A$. I.e., there exists a bisimulation relation R between states such that any transition from $\mathcal{PA}(c)$ can be matched by a transition in A leading to states in R (and its dual for transitions from A). For simplicity, we ignore all reflexive transitions with empty sets as labels in $\mathcal{PA}(c)$, which must exist for all primitive connectors – because the Port Automata semantics assumes that connectors can decide not to have dataflow and remain in the same state.
We show that this definition is correct using an inductive argument. We show that (1) the encodings of primitive channels from Section 2 are correct with respect to the automata from Section 3, and (2) the encoding of a connector built with the sequential, parallel, or trace operators is correct with respect to the automata of their parts after composing the appropriate ports. Note that γ and id_n are regarded here as primitive connectors.

Lemma 1 (Correctness of primitive's encodings). *Any primitive from Fig. 4 is correct w.r.t. its corresponding automaton from Fig. 5, after renaming ports in the latter to follow the same convention as in the encoding (e.g., $(1, \mathsf{sr})$ instead of a).*

Proof. We will only show that this lemma holds for one of the connectors, the fifo, because the other connectors can be shown in a similar way. Recall that after Definition 3 we defined $\mathcal{PA}(\mathsf{fifo})$ as an example. The resulting automaton has 4 transitions, and after ignoring the reflexive and empty transitions only two remain. Recall also the port automaton of the fifo in Fig. 5. It is enough to observe that $R = \{\langle \mathsf{fifo}, q_0 \rangle, \langle \mathsf{fifofull}, q_1 \rangle\}$ is a strong bisimulation between the two automata, after replacing a by $(1, \mathsf{sr})$ and b by $(1, \mathsf{sk})$.

Lemma 2 (Correctness of $\mathcal{PA}(c_1; c_2)$). *If $\mathcal{PA}(c_1)$ and $\mathcal{PA}(c_2)$ are correct with respect to A_1 and A_2, respectively, and $c_1; c_2$ is well-typed, then $\mathcal{PA}(c_1; c_2)$ is correct with respect to $(A_1\sigma_1 \bowtie A_2\sigma_2)\backslash X$, where σ_1,*

σ_2 and X define port renamings and hiding of ports that mimic the connecting of ports from c_1 to c_2:

$$\begin{array}{l}\sigma_1 = \{(i, \mathsf{sk}) \mapsto (i, \mathsf{mx}) \mid (i, \mathsf{sk}) \in N_1\} \\ \sigma_2 = \{(i, \mathsf{sr}) \mapsto (i, \mathsf{mx}) \mid (i, \mathsf{sr}) \in N_2\}\end{array} \quad X = \{(i, \mathsf{mx}) \mid (i, \mathsf{sk}) \in N_1\}$$

Proof. We provide only a sketch of the proof. This proof follows in two phases. First, by considering a transition $(p, q) \xrightarrow{K} (p', q')$ in $(A_1\sigma_1 \bowtie A_2\sigma_2)\backslash X$, one can conclude by performing a case analysis that $\exists (p; q) \xrightarrow{K'} (p'; q')$ in $\mathcal{PA}(c_1; c_2)$. Second, by verifying that the dual also holds.

Lemma 3 (Correctness of $\mathcal{PA}(c_1 \oplus c_2)$). *If, for $i \in \{1, 2\}$, $\mathcal{PA}(c_i)$ is correct with respect to $A_i = (Q_i, N_i, \rightarrow_i, q_{0,i})$, $c_i : n_i \rightarrow m_i$, and $c_1 \oplus c_2$ is well-typed, then $\mathcal{PA}(c_1 \oplus c_2)$ is correct with respect to $A_1 \bowtie (A_2\sigma)$, where σ defines port renamings:*

$$\begin{array}{rl}\sigma = & \{(i, \mathsf{sr}) \mapsto (i + n_1, \mathsf{sr}) \mid (i, \mathsf{sr}) \in N_2\} \\ \cup & \{(j, \mathsf{sk}) \mapsto (j + m_1, \mathsf{sr}) \mid (j, \mathsf{sk}) \in N_2\}\end{array}$$

Proof. We provide only a sketch of the proof. This proof follows the same strategy as the proof for the sequential composition. Start by considering a transition $(p, q) \xrightarrow{K} (p', q')$ in $(A_1 \bowtie A_2)\sigma$. By analysing the possible cases, it is possible to conclude that $\exists (p \oplus q) \xrightarrow{K'} (p' \oplus q')$ in $\mathcal{PA}(c_1 \oplus c_2)$ that mimics this transition. A similar argument for its dual can also be made.

Theorem 1 (Correctness of $\mathcal{PA}$). *Given a well-typed connector c, $\mathcal{PA}(c)$ is correct with respect to some port automaton A built by composing the automata of the primitive connectors within c.*

Proof. This result follows by induction on the structure of connectors, whereas the base case is captured by Lemma 1, and the inductive steps are captured by Lemmas 2 and 3, and by the fact that the trace operation can also be shown correct with respect to some port automaton – due to space restrictions, and because the proof follows similar steps to Lemma 2, we omit here that proof.

4 Connectors as mCRL2 Specifications

The mCRL2 toolset consists of a collection of tools to analyse systems specified in a dedicated process algebra of communicating processes. In a given mCRL2 model, the atomic element of processes are actions. By defining and combining actions we create processes. We describe the core subset of the mCRL2 specification language, focusing on the relevant constructs to understand the encoding of our calculus to mCRL2. A process can be one of the following.

- $a_1|\ldots|a_n$. P – *atomic execution* of n actions (a_1 until a_n), where $n \geq 1$, followed by the execution of P;

id_1	$Id1 = a\|b \;.\; Id1$	lossy	$Lossy = (a + a\|b) \;.\; Lossy$
fifo	$Fifo = a \;.\; b \;.\; Fifo$	Δ_2	$Dupl = a\|b\|c \;.\; Dupl$
drain	$Drain = a\|b \;.\; Drain$	∇_2	$Merger = (a\|c + b\|c) \;.\; Merger$

Table 1: mCRL2 processes of primitives, for some actions a, b, c.

- $P + Q$ – *non-deterministic choice* between two processes P and Q;
- $P \parallel Q$ – *parallel execution* of a process P and a process Q (interleaved or at the same time);
- $\delta_H(P)$ – *encapsulation*, blocking the actions in H when executing P;
- $\Gamma_C(P)$ – *communication* of ports, where C is a mapping from groups of atomic actions $a_1|\ldots|a_n$ to another action b (with $n \geq 2$), replacing all groups of actions $a_1|\ldots|a_n$ by b in the execution of P.
- *Reference* to a process name P defined in the scope of the process.

An *mCRL2 program* consists of a pair (P, π) with a process P and a mapping π from process names to process definitions (with possibly recursive definitions), as described above.
The full language is rich enough to capture aspects such as data types and parametrised actions, which we do not explore here. Given a specification in mCRL2 one can, for example, compile and visualise its corresponding labelled transition system, and can verify properties in a dedicated dynamic calculus with fix points.

4.1 Encoding CRC into mCRL2 programs

We adapt the translation by Kokash et al. [7]. Although the authors encode different connector semantics into mCRL2 programs, we focus on their encoding into constraint automata, for which they have a correctness proof (which ignores data constraints, similarly to CRC).
Table 1 presents the mCRL2 process definitions for the primitives used in Fig. 5. These can be combined in parallel to produce more complex connectors, as exemplified below.

Example 1. Consider the connector $c = \mathsf{id}_1; \Delta_2; (\mathsf{fifo} \oplus \mathsf{lossy})$. Each channel in the connector maps to the following processes:

$$Id_1 = (a|b) \;.\; Id_1 \qquad Fifo = f \;.\; g \;.\; Fifo$$
$$Dupl = c|d|e \;.\; Dupl \qquad Lossy = (h + h|i) \;.\; Lossy$$

Let π_c be the set of definitions above. A program for c can be built by placing these definitions in parallel, by imposing communication with Γ, and by encapsulating internal ports with δ. For example, the program (P_c, π_c), with P_c defined below, provides a (naive) encoding of the behaviour of c, which only exposes the ports a, g, and i.

$$P_c = \delta_{\{b,c,d,e,f,h\}} \\ (\Gamma_{\{b|c \to bc, d|f \to df, e|h \to eh\}}(Sync \parallel Dupl \parallel Fifo \parallel Lossy))$$

This naive approach to combine connectors leads to an exponential increase of combinations of actions as the connector grows, which quickly becomes untreatable by the mCRL2 tools. This problem is addressed by performing communication and encapsulation as soon as possible, i.e., everytime a new primitive is connected to a connector [7]. Our encoding follows the same ideas, performing encapsulation as soon as possible.

Definition 4 (Encoding $\mathcal{MC}$). *The encoding $\mathcal{MC}$ follows a similar approach to $\mathcal{PA}$, where actions follow the pattern $(n, \mathsf{sr})_\ell$, $(n, \mathsf{sk})_\ell$, or $(n, \mathsf{mx})_\ell$ to indicate that n-th source, sink, or mixed port, using the unique identifier ℓ to distinguish between actions from different basic automata. We start by defining auxiliary functions Block, Hide, and Com, used to describe ports that are blocked, are hidden, and communicate. $N_{i,\ell}$ is the name we give to processes denoting nodes that connect pairs of ports.*

$$\mathsf{Block}(n, \ell_1, \ell_2) = \bigcup_{1 \le i \le n} \{(i, \mathsf{sk})_{\ell_1}, (i, \mathsf{sr})_{\ell_2}\} \quad \mathsf{Hide}(n, \ell_1) = \bigcup_{1 \le i \le n} \{(i, \mathsf{mx})_{\ell_1}\}$$

$$\mathsf{Com}(n, \ell_1, \ell_2, \ell) = \bigcup_{1 \le i \le n} \{(i, \mathsf{sk})_{\ell_1} | (i, \mathsf{sr})_{\ell_2} \to (i, \mathsf{mx})_\ell\}$$

Given a connector c and a unique identifier ℓ, $\mathcal{MC}(c)_\ell$ is defined below.

$$\begin{aligned}
\mathcal{MC}(p)_\ell &= (P_\ell, \{P_\ell = \mathsf{Primitive}(p, \ell)\}) \\
&\quad \textit{where } \mathsf{Primitive}(p, \ell) \textit{ is the process of primitive } p \textit{ (c.f. Table 1),} \\
&\qquad \textit{using the proposed notation for actions marked by } \ell. \\
\mathcal{MC}(c_1; c_2)_\ell &= (P_\ell, \{P_\ell = \tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}(n,\ell_1,\ell_2)}(\Gamma_{\mathsf{Com}(n,\ell_1,\ell_2,\ell)} \\
&\qquad (P_1 \| P_2)))\} \cup \pi_1 \cup \pi_2) \\
&\quad \textit{where } c_1 : n_1 \to n \quad c_2 : n \to n_2 \\
&\qquad (P_1, \pi_1) = \mathcal{MC}(c_1)_{\ell_1} \quad (\ell_1 \textit{ is fresh}) \\
&\qquad (P_2, \pi_2) = \mathcal{MC}(c_2)_{\ell_2} \quad (\ell_2 \textit{ is fresh}) \\
\mathcal{MC}(c_1 \oplus c_2)_\ell &= (P_\ell \ , \ \{P_\ell = (P_1 \parallel P_2)\} \cup \pi_1 \cup \pi_2') \\
&\quad \textit{where } c_1 : n_1 \to m_1 \quad c_2 : n_2 \to m_2 \\
&\qquad (P_1, \pi_1) = \mathcal{MC}(c_1)_\ell \\
&\qquad (P_2, \pi_2) = \mathcal{MC}(c_2)_{\ell_2} \quad (\ell_2 \textit{ is fresh}) \\
&\qquad \pi_2' = \pi_2 \{(i, \mathsf{sr})_{\ell_2} \mapsto (i + n_1, \mathsf{sr})_\ell \mid 1 \le i \le n_2\} \cup \\
&\qquad\qquad \{(j, \mathsf{sk})_{\ell_2} \mapsto (j + m_1, \mathsf{sk})_\ell \mid 1 \le j \le m_2\}
\end{aligned}$$

The definition of $\mathcal{MC}(\mathsf{Tr}_n(c))_\ell$ is omitted here, and follows a similar structure to the encoding of $\mathcal{MC}(c_1; c_2)_\ell$.

We illustrate this encoding using a simplified version of Example 1.

Example 2. Let $x = \Delta_2;(\mathsf{fifo} \oplus \mathsf{lossy})$ and a, b, c, d, e be unique identifier:

$$
\begin{aligned}
\mathcal{MC}(\mathsf{fifo} \oplus \mathsf{lossy})_a &= (P_a, \pi_a) \\
\pi_a &= \{P_a = \mathit{Fifo}_a \parallel \mathit{Lossy}_a \\
&\quad , \mathit{Fifo}_a = (1,\mathsf{sr})_a|(1,\mathsf{sk})_a \ . \ \mathit{Fifo}_a \\
&\quad , \mathit{Lossy}_a = ((2,\mathsf{sr})_a + (2,\mathsf{sr})_a|(2,\mathsf{sk})_a) \ . \ \mathit{Lossy}_a\} \\
\mathcal{MC}(x)_b &= (P_b, \pi_b) \\
\pi_b &= \{P_b = \tau_{\{(1,\mathsf{mx})_b,(2,\mathsf{mx})_b\}}(\delta_{\{(1,\mathsf{sk})_c,(1,\mathsf{sr})_a,(2,\mathsf{sk})_c,(2,\mathsf{sr})_a\}} \\
&\quad (\Gamma_{\{(1,\mathsf{sk})_c|(1,\mathsf{sr})_a \to (1,\mathsf{mx})_b,(2,\mathsf{sk})_c|(2,\mathsf{sr})_a \to (2,\mathsf{mx})_b)\}} \\
&\quad\quad (\Delta_{2,c} \| P_a))) \\
&\quad , \ \Delta_{2,c} = (1,\mathsf{sr})_c|(1,\mathsf{sk})_c|(2,\mathsf{sk})_c \ . \ \Delta_{2,c}\} \cup \pi_a
\end{aligned}
$$

4.2 Correctness of $\mathcal{MC}_\ell(\cdot)$

Kokash et al. [7] have shown the correctness of a similar encoding from Port Automata (which they call data-agnostic Constraint Automata) to mCRL2. We claim that the correctness of our encoding follows from the correctness of CRC with respect to Port Automata, and from the correctness by Kokash et al. regarding mCRL2 specifications, as depicted in Fig. 6. We defined the encoding $\mathcal{MC}$ from CRC—and not from the PA model—to preserve the parallel structure of the communicating components, which would be lost if our starting point would be the (flatten) tile semantics followed by the $\mathcal{PA}$ encoding.

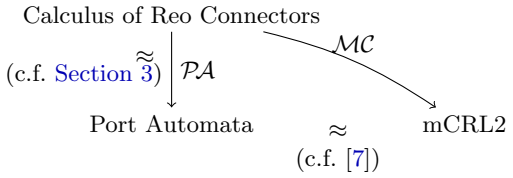


Fig. 6: Relation between CRC, PA, and mCRL2.

Fig. 6 highlights the two correctness results, via bisimulations, between the connector calculus, the PA semantcs, and mCRL2 programs. Note that we do not formally show that our encoding matches precisely the encoding from Kokash et al. [7], and only explain that our encoding follows the same ideas as the previous encoding to mCRL2.

5 ReoLive framework

The ReoLive framework combines tools that analyse connectors and families of connectors under a single web-based front-end. The project

and a compiled snapshot can be found online in https://github.com/ReoLanguage/ReoLive. This section focuses on what the framework currently offers, and gives less how to extend it with new plug-ins. More concretely, it describes how to specify connectors and how to visualise it and analyse it using the Port Automata and the mCRL2 encodings.

5.1 Architecture

This project combines software artefacts in more than one programming languages. The core tools to parse and analyse connectors are implemented in Scala by the Preo project,[1] which is either compiled into JavaScript, using the Scala.js compiler,[2] or into a client-server pair of programs. In the latter, the client is compiled also into JavaScript and the server is based on the Play framework,[3] and is compiled into Java binaries. Furthermore, both JavaScript programs use the D3 JavaScript libraries[4] to produce the graph layouts, which manipulate SVG-based diagrams.

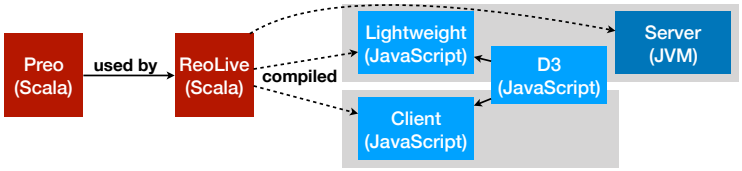


Fig. 7: Architecture of the ReoLive implementation.

The overall architecture is summarised in Fig. 7. The code can be compiled in two different ways: by producing a *standalone* JavaScript library (bottom right rectangle), or by producing a *client-server* architecture (top right rectangle). The former has the advantage of being easier to distribute (a snapshot of our implementation can be found online), while the latter has the advantage of being more powerful and complete (currently using an SMT solver for more complex families of connectors, but the server has to be compiled and executed locally). The ReoLive project website keeps a snapshot of a recent version of the standalone version, depicted in Fig. 8. This web front-end is subdivided into different containers we call *widgets*: (1) where the user specifies connectors, (2) displays the connector's type, (3) displays a concrete instance and its type, (4) presents example connectors to help knowing the syntax, (5) depicts graphically the instance from 3, (6) depicts the Port Automata of that instance (c.f. Section 3), and (7) outputs the mCRL2 program (c.f. Section 4), ready to be analysed by mCRL2 tools.

[1] https://github.com/ReoLanguage/Preo
[2] https://www.scala-js.org
[3] https://www.playframework.com
[4] https://d3js.org

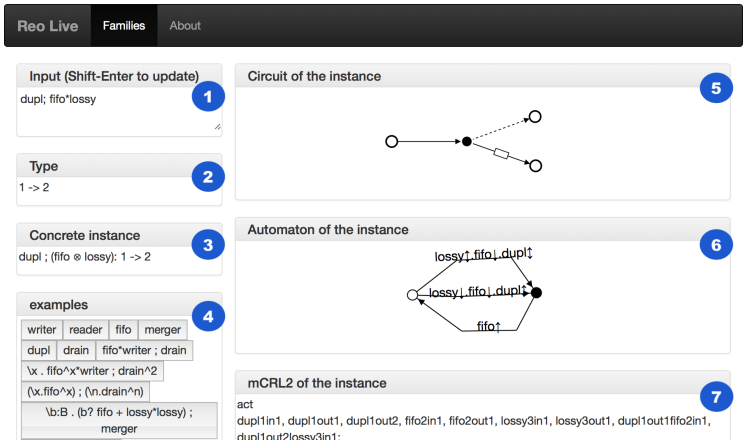


Fig. 8: Screenshot of the standalone version of ReoLive's website.

5.2 The Preo Language

The Preo language is a concrete language for the calculus described in [9], given by the grammar below.

$$c = p \in \mathcal{P} \mid \texttt{id} \mid \texttt{sym}(n_1, n_2) \mid c;c' \mid c*c' \mid \texttt{Tr}(n)(c) \mid c\hat{}n \mid \ldots$$

The rest of the syntax, corresponding to the ellipsis, concern families of connectors, i.e., how to define and restrict parameters that, once instantiated, lead to different connectors of a same family. This is out of the scope of this paper. Furthermore, the language includes the `reader` and `writer` constructs, describing Reo reader and writer components, and supports the definition of named *subconnectors*. The set of primitives $\mathcal{P}$ include mergers, duplicators, fifo channels, lossy channels, and synchronous drains, but others can be easily included. The complete list of primitives can be found by exploring the examples in widget (4) from Fig. 8. Our running example in widget (1) "`dupl;fifo*lossy`" corresponds to the connector $\Delta_2; (\mathsf{fifo} \oplus \mathsf{lossy})$, also used in Example 2.

5.3 Interconnecting widgets

The content of the website is subdivided into *widgets*, as highlighted in Fig. 8. Internally a widget is a statefull object that can interact with the user, and produces a value when *fired*, possibly using values produced by other widgets. Each widget defines its own firing behaviour: the *Type* widget (2) produces a typed connector and its type from the value of the *Input* widget (1); the *Instance* widget (3) calculates a concrete instance *con* based on the connector from the *Type* widget; the *Circuit* widget (5) calculates and depicts a graph structure of *con* in the *Instance* widget; the *Automaton* widget (6) calculates and depicts a Port Automaton of *con*, as described in Section 3; and the *mCRL2* widget (7) calculates and displays the mCRL2 program of *con*, as described in Section 4.

A special event may trigger a sequence of firings—in our case, pressing shift-enter triggers the firing of all widgets in order except widget (4). More complex orchestration mechanisms of widgets, based on the concept of reactive programming, are left for future work. Furthermore, widgets can be *active* or *inactive*; to toggle between these one only needs to press the header of the widget, and when inactive the content of the widget is not displayed. Only active widgets are fired, and when a widget is fired when it becomes active. In the client-server architecture widgets can further possess a callback function with a dedicated firing behaviour triggered by the server.

Example 3. We use the example in Fig. 8 to guide a more detailed explanation of each widget. The user starts by specifying the connector "`dupl;fifo*lossy`" in the Input widget. When pressing Shift-Enter, the Input Widget stores the string internally, so that other widgets can access it. The Type widget accesses this string, parses it, produces the connector $\Delta_2; \mathsf{fifo} \oplus \mathsf{lossy}$, and type checks it. The resulting type $1 \mapsto 2$ is depicted, and the connector is stored and made available to other widgets.
The Instance widget simplifies this connector, removing some syntactic sugar – if the connector had parameters, not addressed in this paper, it would search for valid assignments for this parameters, replacing them by the assignment found. This widget then stores and displays the simplified connector alongside its type. In the client-server architecture the Type and Instance widgets are combined: the server receives the string from the Input widget, producing both a type and an instance and sending this information to the corresponding widget.
All the 3 right widgets access the connector stored in the Instance widget. The Circuit widget generates a graph containing a Reo representation of this connector. Some simplifications from the original connector are made, e.g., removing redundant Sync channels, or combining nested mergers into a single merger. The Automaton widget depicts the associated port automaton, using the rules explained in Section 3 to generate the automaton. This widget uses an abstraction of the names for readability, using only the name of the primitives they refer to enhanced with a downward arrow depicting the entry of data into the primitive, an upward arrow depicting data leaving the primitive, and a double arrow to depict both cases. Finally, the mCRL2 widget contains the mCRL2 model of the connector, following the encoding from Section 4. In this model each action is identified by the name and a unique identifier of the primitive it refers to, as well as information about the type of port. For example, the action `fifo2in1` refers to the first source (input) port of the fifo, and 2 is the unique identifier of that fifo primitive connector.

5.4 Towards verification of connector families

The Preo language, as well as the full version of the connector calculus from Proença and Clarke [9], describe families of connectors. In this paper we did not consider the families aspect, although the existing tools to type-check Preo connectors are included in ReoLive.

We experimented on how to verify the full calculus of connector families using the mCRL2 toolset, following the ideas from Beek and de Vink [3]. Unfortunately, mCRL2 requires the number of processes running in parallel to be fixed and known upfront, limiting the analysis to only a bounded set of families. The latest experiments consist of generating a small number of instances of a connector and include them in a single mCRL2 model, which can be used for model checking. However, we did not find a satisfactory approach to either select an interesting set of candidates for instances, or to give some control over the instances being selected. Furthermore, modelling families of connectors can easily produce a state explosion that is hard to control. Hence we left these experiments out of the existing framework, although they can be found in experimental branches on our GitHub project. Future work will involve providing some control over the instances that could be of interest when analysing families of connectors, and investigating a suitable (modal) logic to describe properties over families of connectors.

6 Conclusion and Future Work

This paper describes a semantic model for the connector calculus using the port automata. Based on this model we encode our connector calculus into mCRL2, following Kokash et al. [7]. These two encodings are included in the ReoLive framework, animating our calculus with our web framework which implements the calculus, the port automata semantics, and the mCRL2 of each connector.

Our future work is many-fold. We expect to extend the portfolio of available modules; for example, add support for a dedicated modal logic to verify connectors, analyse different semantics of Reo connectors other than Port Automata (incorporation of the IFTA tools is planned soon),[5] and add support for the Treo language to specify connectors.[6]

Orthogonally, we also plan to improve the client-server version of ReoLive, by taking advantage of the server capabilities. For example, we plan on automatically processing the mCRL2 model encoded, which the user may download, or use to verify the dedicated modal logic for connectors.

Acknowledgements

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within projects POCI-01-0145-FEDER-016692 (first author) and POCI-01-0145-FEDER-029946 (second author).

[5] https://github.com/haslab/ifta
[6] https://github.com/ReoLanguage/Reo

References

1. Farhad Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, June 2004.
2. Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.
3. Maurice H. ter Beek and Erik P. de Vink. Using mcrl2 for the analysis of software product lines. In *Proceedings of the 2Nd FME Workshop on Formal Methods in Software Engineering*, FormaliSE 2014, pages 31–37, New York, NY, USA, 2014. ACM.
4. Fabio Gadducci and Ugo Montanari. The tile model. In *Proof, Language, and Interaction*, pages 133–166. MIT Press, 2000.
5. Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck van Weerdenburg. The formal specification language mCRL2. In *Methods for Modelling Software Systems (MMOSS)*, Dagstuhl Seminar Proceedings, 2007.
6. Christian Koehler and Dave Clarke. Decomposing port automata. In *Proc. SAC '09*, pages 1369–1373, New York, NY, USA, 2009. ACM.
7. Natallia Kokash, Christian Krause, and Erik P. de Vink. Reo + mCRL2: A framework for model-checking dataflow in service compositions. *FAC*, 24(2):187–216, 2012.
8. Saunders MacLane. Categorical algebra. *Bull. Amer. Math. Soc.*, 71(1):40–106, 01 1965.
9. José Proença and Dave Clarke. Typed connector families and their semantics. *Sci. Comput. Program.*, 146:28–49, 2017.