A Uniform Framework for Modeling and Verifying Components and Connectors

Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz*

Faculty of Computer Science,
Technische Universität Dresden, Germany

Abstract. The purpose of this paper is to present a framework to model component interfaces and the component connectors that provide the glue code for the components. Our modeling approach is based on two input languages which rely on the same automata model. One of them is a scripting language which can serve to specify exogenous or endogenous coordination mechanisms. The other one is a guarded command language which has been designed to specify behavioral component interfaces, but can also be used to design component connectors. This hybrid approach allows nesting of the two specification languages, supports compositional design, modular verification and reusability of components or component connectors. It yields the input language of our verification toolset Vereofy which realizes several model checking algorithms for components, component connectors, and the composite system.

1 Introduction

The basic principle of component-based software engineering is to fragment a complex system into logical components with well-defined interfaces. In this context, a variety of coordination models and languages have been introduced that support the separation between computations inside the components and the interactions between the components. *Endogenous* coordination languages require to incorporate coordination primitives within the code that specifies the behavior of the components. A typical example is Linda [11] where components are described in a computational languages extended by operators to store or retrieve data objects from a global tuple space. A cleaner separation of computation and coordination is provided by *exogenous* coordination models where the components do not need to be aware of each other. Instead, they are controlled from "outside" via their interfaces. Several approaches for exogenous coordination have been suggested, e.g., an aspect-oriented approach [8], a variant of the π-calculus with anonymous peer-to-peer communication [13], and formalisms that rely on the construction of component connectors, such as interaction systems as in [12,17] or the declarative channel-based language Reo [2].

For providing tool support for the verification of systems specified in such coordination formalisms, one needs input languages which on the one hand cover the

* The authors are supported by the EU-project Credo and the DFG-project Syanco.

J. Field and V.T. Vasconcelos (Eds.): COORDINATION 2009, LNCS 5521, pp. 247–267, 2009.
© Springer-Verlag Berlin Heidelberg 2009

major features of the coordination language and on the other hand have an operational semantics that can easily be implemented. For exogenous coordination languages, there is an additional aspect that should be taken into account. The objects specified in an exogenous coordination language typically just formalize the network that organizes the interactions of black-box components, but they do not make any restrictions on the behavior of the components. Such restrictions, however, might be essential to prove certain functional properties of the composite system consisting of several components and the network that serves as a component connector. Thus, what is needed for model checking tools are input languages that provide coordination primitives to specify the network and features to model the behavioral interfaces of the components. Beside constraints on the type of messages that can be send or received via the input and output ports, such behavioral interfaces can also specify local states of the components and impose (possibly data-dependent) conditions on the enabledness of sending and receiving messages via the I/O-ports. In the literature, there are several *automata-based models* that can be seen as "light-weight" formalisms for specifying the behavioral interfaces of components. Examples are I/O-automata that support compositional reasoning about asynchronous concurrent systems [16] or interface automata that have been introduced for reasoning about compatibility of components [10,9] or constraint automata which have been developed in the context of reasoning about exogenous coordination [5].

The goal of this paper is to present the uniform framework for specifying behavioral component interfaces and component connectors that we developed in the context of our toolkit Vereofy [7] (see Fig. 1). The verification constituents of Vereofy are symbolic BDD-based model checking tools for linear-, branching- and alternating-time temporal logics with special operators to reason about the data flow at I/O-ports of components or internal nodes of the network and a bisimulation checker. These logics LTL_{IO}, BTSL, ASL and verification algorithms for them and the bisimulation checking algorithm, together with some experimental results performed on the basis of a prototype implementation, have been presented elsewhere [6,15,4,14]. This work on model checking component-based systems uses *constraint automata* [5] as a uniform operational model for component connectors, behavioral component interfaces and the composite system. The focus of this paper is on the modeling approach of Vereofy which supports

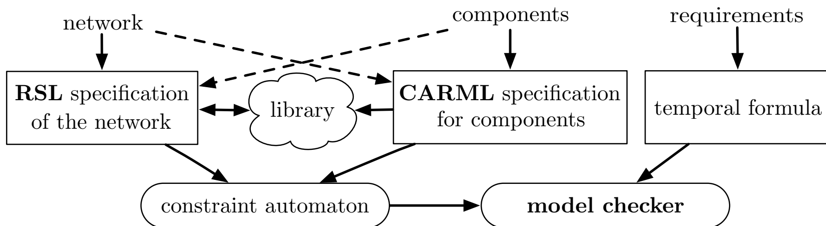


Fig. 1. Vereofy overview

exogenous and endogenous coordination. Vereofy deals with the combination of two languages, both equipped with a constraint automata semantics. One of them is a guarded command language, called CARML, that mainly serves to specify the I/O-ports of components and their stepwise "observable" behavior. The second input language of Vereofy, called RSL, is a *scripting language* which combines the major features of the exogenous coordination language Reo [2] with concepts to specify connectors with dynamically changing network topologies and some features of other languages (such as shared variables or data types). Reo's coordination primitives allow to reason about all kinds of coordination patterns with an arbitrary mixture of synchronous or asynchronous peer-to-peer communication. By combining Reo's coordination primitives with operators for the instantiation of component connectors or components which are specified in RSL or CARML, our approach offers an elegant way for the compositional, hierarchical construction of component connectors and components. While previous work on Reo and constraint automata relies on the assumption of a global data domain that serves as data type for all messages that can be send via the channels, our hybrid modeling approach with RSL and CARML supports the use of several data types. This allows using standard methods to ensure type consistency and to check compatibility via type checking.

Organization of the paper. Section 2 summarizes our notations concerning data types and constraint automata. The languages CARML and RSL are presented in Section 3 and 4, respectively. An example using our hybrid modeling approach is provided in Section 5 and experimental results with our toolkit Vereofy are presented in Section 6. The paper ends with some remarks on related work and a conclusion (Section 7).

2 Preliminaries

Data types with fixed semantics. Let DT denote a set of data types covering standard data types, such as Booleans or integers of fixed bit-size, or user-defined data types with a fixed semantics, such as arrays and unions over elements of a predefined data type or enumeration types. Let Op denote a set of operators on the data types in DT such as conjunction, disjunction, negation for Booleans and arithmetic operations like addition and multiplication for integers. Furthermore, Pred denotes a set of predicates, such as the standard binary comparison predicates $=, <, \leq$, and so on. Formally, the elements of DT are fixed finite nonempty sets and each operator $\mathsf{op} \in \mathsf{Op}$ is a function $\mathsf{op} : \mathsf{T}_1 \times \ldots \times \mathsf{T}_k \rightarrow \mathsf{T}$ where $(\mathsf{T}_1, \ldots, \mathsf{T}_k, \mathsf{T}) \in \mathsf{DT}^{k+1}$ and $k \in \mathbb{N}$. The tuple $(\mathsf{T}_1, \ldots, \mathsf{T}_k, \mathsf{T})$ is called the type of op, denoted $\mathfrak{type}(\mathsf{op})$. Each predicate $\mathsf{P} \in \mathsf{Pred}$ is a subset of $\mathsf{T}_1 \times \ldots \times \mathsf{T}_k$ where $k \geq 1$ and $\mathsf{T}_1, \ldots, \mathsf{T}_k \in \mathsf{DT}$. We write $\mathfrak{type}(\mathsf{P})$ for the tuple $(\mathsf{T}_1, \ldots, \mathsf{T}_k)$.

Uninterpreted data types and signatures. Beside data types with fixed semantics, our languages also allow for uninterpreted symbols for data types, operators and predicates. These can be used as parameters for RSL and CARML specifications which then serve as templates for components or connectors.

A signature is a tuple $\mathfrak{Sig} = (\mathfrak{DT}, \mathfrak{Op}, \mathfrak{Pred}, \mathfrak{Var})$ where $\mathsf{DT} \subseteq \mathfrak{DT}$, $\mathsf{Op} \subseteq \mathfrak{Op}$ $\mathsf{Pred} \subseteq \mathfrak{Pred}$ and $\mathfrak{Var}$ is a set of typed variables, i.e., for each variable $V \in \mathfrak{Var}$ its type $\mathfrak{type}(V)$ is an element of $\mathfrak{DT}$. The uninterpreted symbols for data types, operators and predicates are then given by the elements of $\Theta = \mathfrak{DT} \setminus \mathsf{DT}$, $\Omega = \mathfrak{Op} \setminus \mathsf{Op}$ and $\Pi = \mathfrak{Pred} \setminus \mathsf{Pred}$, respectively. The data type symbols $T \in \Theta$ can be seen as placeholders for sets (data types). The operator symbols $op \in \Omega$ and predicate symbols $P \in \Pi$ are associated with a type. The type of an operator symbol $op \in \Omega$ is a tuple $\mathfrak{type}(op) = (T_1, \ldots, T_k, T) \in \mathfrak{DT}^{k+1}$ for some $k \in \mathbb{N}$. It declares that op takes as arguments k elements $v_1, \ldots, v_k$ where v_i is of type T_i and returns an element of type T. That is, op stands for a function $op : T_1 \times \ldots \times T_k \to T$. The number k is called the arity of op. Similarly, the type of a predicate symbol $P \in \Pi$ is a tuple $\mathfrak{type}(P) = (T_1, \ldots, T_k) \in \mathfrak{DT}^k$ for some $k \geq 1$, denoting that P has to be interpreted by a predicate consisting of tuples $(v_1, \ldots, v_k)$ where v_i is an element of data type T_i.

Terms and atomic propositions. Terms over $\mathfrak{Sig}$ are built by variables, constants, and the operator symbols in a type-consistent manner. Formally, terms over $\mathfrak{Sig}$ are defined recursively according to the following statements.

(1) Each variable $V \in \mathfrak{Var}$ is a term of type $\mathfrak{type}(V)$ and each constant $op \in \mathfrak{Op}$ (i.e., 0-ary operator in $\mathsf{Op} \cup \Omega$) is a term of type $\mathfrak{type}(op)$.
(2) If $t_1, \ldots, t_k$ are terms such that the type of t_i is T_i and $op \in \mathfrak{Op}$ with $\mathfrak{type}(op) = (T_1, \ldots, T_k, T)$ then $op(t_1, \ldots, t_k)$ is a term of type T.

Atomic propositions over $\mathfrak{Sig}$ are type-consistent expressions stating that a certain tuple of terms is an element of a predicate in $\mathfrak{Pred}$. Formally, if $P \in \mathfrak{Pred}$ with $\mathfrak{type}(P) = (T_1, \ldots, T_k)$ and $t_1, \ldots, t_k$ are terms over $\mathfrak{Sig}$ such that t_i is of type T_i then $P(t_1, \ldots, t_k)$ is called an atomic proposition over $\mathfrak{Sig}$.

Interpretations of signatures. For the semantics of terms and atomic propositions over a signature, we have to consider an interpretation $\mathcal{I}$ that provides type-consistent meanings for the uninterpreted data types, operator and predicate symbols and the variables. That is, $\mathcal{I}$ assigns a finite set $T^{\mathcal{I}} \neq \emptyset$ to each data type symbol T, an element of $V^{\mathcal{I}} \in T^{\mathcal{I}}$ to each variable of type T, a function $op^{\mathcal{I}} : T_1^{\mathcal{I}} \times \ldots T_k^{\mathcal{I}} \to T^{\mathcal{I}}$ to each operator symbol op of type $(T_1, \ldots, T_k, T)$ and a predicate $P^{\mathcal{I}} \subseteq T_1^{\mathcal{I}} \times \ldots T_k^{\mathcal{I}}$ to each predicate symbol P of type $(T_1, \ldots, T_k)$. (We treat $\mathcal{I}$ as an interpretation for all symbols of $\mathfrak{Sig}$ by putting $\mathsf{S}^{\mathcal{I}} = \mathsf{S}$ for all predefined symbols $\mathsf{S} \in \mathsf{DT} \cup \mathsf{Op} \cup \mathsf{Pred}$.) The semantics $t^{\mathcal{I}} \in T^{\mathcal{I}}$ of a term of type T and the truth value $P(t_1, \ldots, t_k)^{\mathcal{I}} \in \{\text{true}, \text{false}\}$ for an interpretation $\mathcal{I}$ and an atomic proposition $P(t_1, \ldots, t_k)$ are defined in the obvious way.

Locations and data-flow vocabulary. Locations are points in the network where data flow is observable, e.g., the I/O-ports of components or nodes of the network that serve as a router. In the sequel, $\mathcal{L}$ denotes a finite set of locations. The sets $\mathcal{L}^{src}$ and $\mathcal{L}^{snk}$ are disjoint subsets of $\mathcal{L}$ representing different kinds of locations. Intuitively, $\mathcal{L}^{src}$ stands for the set of input ports (sources), $\mathcal{L}^{snk}$ for the set of output ports (sinks). A data-flow vocabulary over a signature $\mathfrak{Sig}$ is

a tuple $\mathfrak{Voc} = \langle \mathcal{L}, \mathcal{L}^{src}, \mathcal{L}^{snk}, \lambda \rangle$ where $\mathcal{L}$ is a set of locations and $\lambda : \mathcal{L} \rightarrow \mathfrak{DT}$ is a function that assigns to each location A its message-type $\lambda(A)$, i.e., the type of data items that can be passed via location A.

Concurrent I/O-operations (CIO) and Constraint automata (CA). Constraint automata [5] serve as a compositional semantics for components and connectors specified in CARML or RSL. For the syntax of CA we slightly depart from [5] and use types for the messages that can be received or sent. The observable data flow is formalized in a CA by means of a concurrent I/O-operation. These can be understood as the potential actions of a component, a connector or as interactions between several components and the connector in the composite system. A concurrent I/O-operation specifies the locations where at some specific time instance data flow is observed simultaneously. In addition, it specifies the data items that are read or written at the I/O-ports of components or transmitted through locations of the connector. Formally, given an interpretation $\mathcal{I}$ for the signature $\mathfrak{Sig}$ then a concurrent I/O-operation over $\mathcal{I}$ is a function c that assigns to each location $A \in \mathcal{L}$ either a data item of type $\lambda(A)^{\mathcal{I}}$ or the special symbol $\perp$ indicating that there is no data flow at A. We write $Obs(c)$ for the set of locations $A \in \mathcal{L}$ such that $c(A) \neq \perp$. Let $CIO_{\mathcal{I}}$, or briefly CIO, denote the set of all concurrent I/O-operations over $\mathcal{I}$. A CA over an interpretation $\mathcal{I}$ for a signature $\mathfrak{Sig}$ is a tuple $\mathcal{A} = (Q, \mathfrak{Voc}, \rightarrow, Q_0)$, where Q is a finite set of states, $\mathfrak{Voc}$ a data-flow vocabulary over $\mathfrak{Sig}$, $\rightarrow \subseteq Q \times CIO_{\mathcal{I}} \times Q$ the transition relation, and $Q_0 \subseteq Q$ the set of initial states. Obviously, the interpretation $\mathcal{I}$ is irrelevant if all message-types have a fixed semantics, i.e, $\lambda(A) \in \mathsf{DT}$ for all $A \in \mathcal{L}$.

3 Constraint Automata Reactive Module Language

One of the input languages of Vereofy is a guarded command language, called CARML (constraint automata reactive module language), that describes the transitions of constraint automata in a symbolic way, i.e., by means of Boolean conditions on the states and the enabled concurrent I/O-operations. CARML provides a convenient way to specify the component interfaces and to provide a high-level description of the operational behavior of components. CARML supports channel-based message passing and communication over shared variables. The latter is irrelevant for exogenous coordination, but can be useful to incorporate the coordination primitives of an endogenous approach e.g. for existing systems where the coordination protocol is given in an imperative language. In this case, modeling the protocol by means of the coordination language can be much harder than providing a CARML specification. CARML is even expressive enough to specify complex component connectors. To ease the automatic translation of CARML specifications into a compact internal BDD-based representation, we adapted some concepts of reactive modules [1] for the syntax of CARML modules.

Standard data types such as Boolean, integers of fixed bit-size, arrays, unions and enumerations together with the usual operators and predicates on them can be used as data types for variables and message-types for I/O-ports. As in

Section 2, these sets of data types, operators and predicates with fixed semantics are denoted by DT, Op and Pred. CARML modules can also use uninterpreted symbols for data types, operators and predicates. That is, a CARML module $\mathcal{M}$ can be parameterized by a set Θ of data types, a set Ω of operator symbols, a set Π of predicate symbols, and a set Υ of variables with types in $\mathfrak{DT} = \mathsf{DT} \cup \Theta$.

$\texttt{MODULE}\ \mathcal{M}\langle \texttt{type} : \Theta, \texttt{op} : \Omega, \texttt{pred} : \Pi, \texttt{var} : \Upsilon \rangle\ \{$

// interface declaration: source ports

$\texttt{in} : T_1^{\texttt{in}}\ A_1;$

$\vdots$ // A_i with message-types $T_i^{\texttt{in}} \in \mathsf{DT} \cup \Theta$

$\texttt{in} : T_k^{\texttt{in}}\ A_r;$

// interface declaration: sink ports

$\texttt{out} : T_1^{\texttt{out}}\ B_1;$

$\vdots$ // B_i with message-types $T_i^{\texttt{out}} \in \mathsf{DT} \cup \Theta$

$\texttt{out} : T_\ell^{\texttt{out}}\ B_s;$

// definition of local variables X_i with data types $T_i^{\texttt{var}} \in \mathsf{DT} \cup \Theta$

// with initial value $init_value_i \in T_i^{\texttt{var}}$ (optional)

$\texttt{var} : T_1^{\texttt{var}}\ X_1\ \texttt{init} := init_value_1;$

$\vdots$

$\texttt{var} : T_\ell^{\texttt{var}}\ X_\ell\ \texttt{init} := init_value_\ell;$

// transition definitions

$state_guards_1 \ -\!\![\ I/O_guards_1\ ]\!\!\rightarrow\ state_assignments_1;$

$\vdots \qquad \vdots \qquad \vdots$

$state_guards_n \ -\!\![\ I/O_guards_n\ ]\!\!\rightarrow\ state_assignments_n;$ $\}$

Fig. 2. Schema of a CARML module

The general schema of a CARML module is shown in Fig. 2. It consists of a (possibly empty) parameter list, the interface declaration where the source and sink ports of a component and its local variables are defined followed by the transition definitions specifying the behavioral interface. The shorthand notation "$\texttt{type} : \Theta, \texttt{op} : \Omega, \texttt{pred} : \Pi$" in Fig. 2 refers to a list where all uninterpreted symbols $S \in \Theta \cup \Omega \cup \Pi$ are encountered together with the corresponding keyword `type`, `op` or `pred` and their types in case of the operator and predicate symbols. Similarly, $\texttt{var} : \Upsilon$ stands short for an enumeration of all variables in Υ together with the keyword `var` and their types. All variables in Υ are passed according to the concept "call-by-value".

Let $\mathcal{M}$ be the name of the CARML module in Fig. 2. The data types with fixed semantics together with the parameters Θ, Ω, Π, Υ and the set $\mathfrak{Var}_{\mathcal{M}}$ of variables that can be used in $\mathcal{M}$ (see below) constitute the signature of $\mathcal{M}$ which is given by $\mathfrak{Sig}_{\mathcal{M}} = (\mathfrak{DT}, \mathfrak{Op}, \mathfrak{Pred}, \mathfrak{Var}_{\mathcal{M}})$ where $\mathfrak{DT} = \mathsf{DT} \cup \Theta$, $\mathfrak{Op} = \mathsf{Op} \cup \Omega$ and $\mathfrak{Pred} = \mathsf{Pred} \cup \Pi$. The variables that can be used in a CARML module $\mathcal{M}$ are *local variables* and the variables in Υ. The local variables together with their

type have to be listed in the declaration part of $\mathcal{M}$. Thus, the CARML module in Fig. 2 has the local variables $X_1, \ldots, X_\ell$. The type of local variable X_i is $\mathfrak{type}(X_i) = T_i^{\mathtt{var}}$ which has to be an element of $\mathfrak{DT}$. The specification of an initial value for the local variables is optional.

With our hybrid modeling approach where CARML specifications can be embedded in the scripting language RSL (see Section 4) a CARML module can also use *global variables* which have to be declared in the RSL (main) program. Let $\mathfrak{Var}_\mathcal{M}$ be the set of all local variables of $\mathcal{M}$, all variables in Υ and all global variables. We refer to the elements of $\mathfrak{Var}_\mathcal{M}$ as *accessible* variables of $\mathcal{M}$. The interface declaration in Fig. 2 induces the data-flow vocabulary $\mathfrak{Voc}_\mathcal{M} = \langle \mathcal{L}_\mathcal{M}, \mathcal{L}_\mathcal{M}^{src}, \mathcal{L}_\mathcal{M}^{snk}, \lambda_\mathcal{M} \rangle$ where $\mathcal{L}_\mathcal{M}^{src} = \{A_1, A_2, \ldots, A_r\}$, $\mathcal{L}_\mathcal{M}^{snk} = \{B_1, B_2, \ldots, B_s\}$, and $\mathcal{L}_\mathcal{M} = \mathcal{L}_\mathcal{M}^{src} \cup \mathcal{L}_\mathcal{M}^{snk}$. The type of source port A_i is $\lambda_\mathcal{M}(A_i) = T_i^{\mathtt{in}}$, while sink port B_j is of type $\lambda_\mathcal{M}(B_j) = T_j^{\mathtt{out}}$. Again, the types $T_i^{\mathtt{in}}$ and $T_j^{\mathtt{out}}$ are elements of $\mathfrak{DT}$. The sets $\mathcal{L}_\mathcal{M}^{snk}$ and $\mathcal{L}_\mathcal{M}^{src}$ are supposed to be disjoint.

Each *transition definition* consists of local conditions on the current state (a state guard), conditions on the concurrent I/O-operations to be fired (an I/O-guard) and the effect of firing such an I/O-operation on the states (formalized by the state assignments). A *state guard* is a (possibly empty) conjunction of atomic propositions $P(t_1, \ldots, t_k)$ over $\mathfrak{Sig}_\mathcal{M}$. An *I/O-guard* is a condition on the observable data flow, formalized by a Boolean combination of atomic propositions over an extended signature $\mathfrak{Sig}_\mathcal{M}^\mathcal{L}$ that allows to reason about the data items that are observable at the locations in $\mathcal{L}$. Formally, $\mathfrak{Sig}_\mathcal{M}^\mathcal{L}$ denotes the signature that results from $\mathfrak{Sig}_\mathcal{M}$ by adding

- a special type $T_{\mathrm{I/O}}$ that serves for a characterization of the I/O-ports,
- a new monadic predicate symbol *active* with $\mathfrak{type}(active) = T_{\mathrm{I/O}}$,
- constant symbols $data_A$ with $\mathfrak{type}(data_A) = \lambda_\mathcal{M}(A)$ for all $A \in \mathcal{L}_\mathcal{M}$.

The special type symbol $T_{\mathrm{I/O}}$ is needed for technical reasons only. (Note that the location-symbol A is of type $T_{\mathrm{I/O}}$, while its message-type is $\lambda_\mathcal{M}(A) = \mathfrak{type}(data_A)$.) For the interpretations $\mathcal{I}$ of $\mathfrak{Sig}_\mathcal{M}^\mathcal{L}$ we require that $T_{\mathrm{I/O}}^\mathcal{I} = \mathcal{L}_\mathcal{M}$. The intuitive meaning of the atomic proposition $active(A)$ is a port activity flag which indicates that data flow at location A is observed. To avoid an overlap with state guards, we require that all atomic propositions $P(t_1, \ldots, t_k)$ in an I/O-guard contain at least one subterm $data_A$ for some $A \in \mathcal{L}_\mathcal{M}$.

A *state assignment* is a (possibly empty) sequence of assignments for accessible variables, i.e., state assignments have the form $V_1 := t_1 \,;\, \ldots \,;\, V_p := t_p$ where $V_1, \ldots, V_p$ are pairwise distinct variables in $\mathfrak{Var}_\mathcal{M}$ and t_j are terms over the extended signature $\mathfrak{Sig}_\mathcal{M}^\mathcal{L}$. Intuitively, when firing a transition via a concurrent I/O-operation c with a state assignment as above as then in the next state the value of the variables V_i agrees with the value of the term t_i under the interpretation given by the current state and the c. Variables $V \in \mathfrak{Var}_\mathcal{M} \setminus \{V_1, \ldots, V_p\}$ keep their value after the transition has been taken.

Example 1 (A railway track). The CARML module in Fig. 3 serves as a prototype definition for a railway track where trains may either pass or stop. It can

be instantiated providing a data type for the trains and an element of that type indicating that there is no train in the track. The component has one source port A and one sink port B. Local variable "stat" keeps track of the status (free or occupied), while local variable "train" serves to remember which train actually stopped on the track when occupied by a train. □

```
MODULE track⟨type : TrainType, var : TrainType no_train⟩{
 in : TrainType A;                    // A is a source with type(data_A) = TrainType
 out : TrainType B;                   // B is a sink with type(data_B) = TrainType
 var : enum{free, occupied} stat := free;
 var : TrainType train := no_train;
 stat = free–[ active(A) ∧ active(B) ∧ data_A = data_B ]→;
 stat = free–[ active(A) ∧ ¬active(B) ]→ stat := occupied ; train := data_A;
 stat = occupied–[ active(B) ∧ ¬active(A) ∧ data_B = train ]→
                                        stat := free ; train := no_train;      }
```

Fig. 3. CARML module for a railway track

Semantics of a CARML module. The intuitive operational meaning of the transition definitions is as follows. Suppose that q is the current state, which means an evaluation of all accessible variables. Then, nondeterministically a concurrent I/O-operation c and one of the transition definition is chosen such that the state guard holds for q and the I/O-guard is fulfilled by c. The next state is then obtained by modifying q according to the state assignments of the chosen transition definition. This intuitive behavior can be formalized by means of constraint automata. As a CARML module $\mathcal{M}$ as in Fig. 2 serves as a template for components (or connectors), the constraint-automata semantics of $\mathcal{M}$ relies on an interpretation $\mathcal{J}$ for all parameters in $\Theta, \Omega, \Pi, \Upsilon$. For all predefined symbols $\mathsf{S} \in \mathsf{DT} \cup \mathsf{Op} \cup \mathsf{Pred}$ we write $\mathsf{S}^{\mathcal{J}} = \mathsf{S}$.

The constraint automaton $\mathcal{A}_{\mathcal{M},\mathcal{J}} = (Q, \mathfrak{Voc}_{\mathcal{M}}, \rightarrow, Q_0)$ over $\mathfrak{DT} = \mathsf{DT} \cup \Theta$ is defined as follows. The data-flow vocabulary of $\mathcal{A}_{\mathcal{M},\mathcal{J}}$ is $\mathfrak{Voc}_{\mathcal{M}}$. The state space of $A_{\mathcal{M},\mathcal{J}}$ is the set Q consisting of all evaluations of the variables that are accessible for $\mathcal{M}$, i.e., Q is the set of functions q that assign to each variable $V \in \mathfrak{Var}_{\mathcal{M}}$ a data item in $q(V)$ in $\mathfrak{type}(V)^{\mathcal{J}}$. The set Q_0 of initial states consists of all $q_0 \in Q$ such that $q_0(X_i) = \mathit{init_value}_i$ for each local variable X_i where an initial value has been specified in the declaration part of $\mathcal{M}$. For all other variables V, $q_0(V)$ is an arbitrary element in $\mathfrak{type}(V)^{\mathcal{J}}$.

The transition relation $\rightarrow$ is defined as follows. Let $q, q' \in Q$ and $c \in \mathit{CIO}_{\mathcal{J}}$. Let $\mathcal{I} = (q, \mathcal{J})$ be the interpretation for $\mathfrak{Sig}_{\mathcal{M}}$ that agrees with q for all variables $V \in \mathfrak{Var}_{\mathcal{M}}$ and assigns to the symbols in the parameter list of $\mathcal{M}$ the same meaning as $\mathcal{J}$. The pair $(\mathcal{I}, c)$ denotes the interpretation for the extended signature $\mathfrak{Sig}^{\mathcal{L}}_{\mathcal{M}}$ that agrees with $\mathcal{I} = (q, \mathcal{J})$ on the symbols in $\mathfrak{Sig}_{\mathcal{M}}$, interprets the type symbol $T_{\mathrm{I/O}}$ by $T^{\mathcal{I},c}_{\mathrm{I/O}} = \mathcal{L}_{\mathcal{M}} = \{A_1, \ldots, A_r, B_1, \ldots, B_s\}$, and assigns $\mathit{active}^{\mathcal{I},c} = \mathit{Obs}(c) = \{D \in \mathcal{L}_{\mathcal{M}} : c(D) \neq \bot\}$ to the activity predicate.

For $D \in Obs(c)$ the interpretation of $data_D$ is the data element $data_D^{\mathcal{I},c} = c(D)$. For $E \in \mathcal{L}_{\mathcal{M}}$ with $c(E) = \bot$, the interpretation $data_E^{\mathcal{I},c}$ is irrelevant.

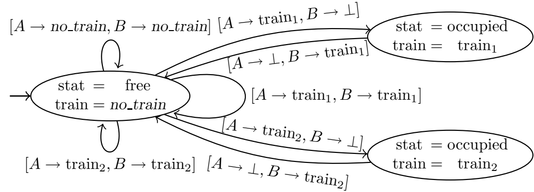
Then, $q \xrightarrow{c} q'$ iff there is a transition definition $state_guards_i$ $-[\ I/O_guards_i\ ]\!\!\rightarrow$ $state_assignments_i$ in $\mathcal{M}$ such that the following conditions (1), (2) and (3) hold.

(1) state q fulfills the state guard, i.e., $state_guards_i$ evaluates to true under $\mathcal{I}$

(2) the concurrent I/O-operation c satisfies the I/O-guard, i.e., I/O_guards_i evaluates to true when interpreted over $(\mathcal{I}, c)$

(3) the next state q' arises from q via executing the state assignments, i.e., if $state_assignments_i$ has the form $V_1 := t_1$; $V_2 := t_2$; ... $V_p := t_p$ then $q'(V_j) = t_j^{\mathcal{I},c}$ for $1 \le j \le p$ and $q(V) = q'(V)$ for $V \in \mathfrak{Var}_{\mathcal{M}} \setminus \{V_1, \ldots, V_p\}$.

The picture on the right shows the reachable part of the CA for the railway track from Example 1 and interpretation $\mathcal{J}$ where $TrainType^{\mathcal{J}}$ is the set $\{\text{train}_1, \text{train}_2, no_train\}$ and $no_train^{\mathcal{J}} = no_train$.



4 Reo Scripting Language

While the main purpose of CARML is to specify the behavioral interfaces of components, Reo scripting language (RSL) mainly serves to specify networks that provide the glue code for components. RSL is inspired by the exogenous coordination language Reo [2] which yields an elegant declarative framework for the compositional construction of connectors by creating channels and glueing their channels ends, the I/O-ports of components, or sub-connectors together. This is done via join-operations, resulting in a network called *Reo circuit*. The semantics of a Reo circuit is a CA which can be constructed in a compositional way by providing constraint automata models for each channel and component and mimicking Reo's operators by corresponding operators for CA [5].

The language RSL combines the Reo operators with operators for the instantiating of components, channels, and component connectors – that are either given by a parameterized CARML module or specified in RSL – and features to specify dynamic reconfigurations of the network topology. Indeed, RSL treats components, channels and component connectors in the same way. This means, that a channel is viewed as a primitive component connector and any component connector can use components or other connectors as "subroutines" via the instantiation mechanism. In what follows, the notion "module" will be used as an umbrella term for component, channel or component connector.

RSL programs. On the top-level, an RSL program consists of (1) a declaration part, (2) a list of include-instructions to access modules from a library, (3) a list of the CARML modules and RSL scripts that might be instantiated in the main

system, and optionally (4) an instantiation of an RSL script that serves as the main system. When omitting (4), a non-parameterized module called “main” is required to be contained in (3). The declaration part of an RSL program contains the declarations of global variables and the definition of user-defined data types (like enumerations, arrays and unions over predefined or previously user-defined data types) as well as constants, operators of arity ≥ 1 and predicates. These together with the predefined data types and built-in operators and predicates constitute the sets DT, Op, Pred of data types, operators and predicates with fixed semantics and only these can be passed as meanings for the uninterpreted symbols (elements of $\Theta, \Omega, \Pi, \Upsilon$) in the parameter list of the instantiated module in (4). This ensures that the instantiation in (4) yields an interpretation for all free symbols in the RSL script for the main system. In (2) the library contains the CARML or RSL code for several predefined basic channel types, such as synchronous channels, FIFO channels, and so on, but also components connectors that serve as coordination units in many situations, like an exrouter or sequencer. It can be extended by user-defined modules.

$$\texttt{CIRCUIT}\ \mathcal{C}\ \langle \texttt{type} : \Theta, \texttt{op} : \Omega, \texttt{pred} : \Pi, \texttt{var} : \Upsilon \rangle\ \{$$

```
    stmt;                      // stepwise construction of a Reo circuit
    interface_decl    // declaration of the I/O-ports of the Reo circuit
}
```

Fig. 4. Schema of an RSL circuit

RSL scripts for networks with static topology. The schema for the RSL script of a Reo circuit without dynamic reconfiguration is shown in Fig. 4 where the parameterization by uninterpreted symbols for data types, operators, predicates and variables is as for CARML modules. The body of an RSL circuit consists of a *statement* that describes the stepwise construction of a Reo circuit and an *interface declaration* where the exported I/O-ports are specified.

Statements. The statement in the body of an RSL circuit is build by basic operations and control flow instructions (sequential composition, conditional branching and for-loops). The abstract syntax for statements is given by the grammar

$$\begin{aligned} \mathit{stmt}\ ::= &\ \mathit{instantiation}\ \big|\ \mathit{Reo_operation}\ \big|\ \mathit{assignment}\ \big|\ \mathit{stmt}\,;\,\mathit{stmt}\ \big| \\ &\ \texttt{if}\ (\mathit{bexpr})\ \{\mathit{stmt}\}\ \texttt{else}\ \{\mathit{stmt}\}\ \big|\ \texttt{for}\ (i = j, \ldots, k)\ \{\mathit{stmt}\} \end{aligned}$$

Instantiation. The instantiation of a module (i.e., a component, a channel or a complex connector) specified in CARML or by a RSL circuit is performed via instructions of the form

$$\texttt{new}\ \ \mathsf{module_template}\langle \Theta', \Omega', \Pi', \Upsilon' \rangle (A_1, \ldots, A_r; B_1, \ldots, B_s)$$

where $\Theta', \Omega', \Pi', \Upsilon'$ are lists of data types, operators, predicates and variables or constants that provide meanings for the parameters in the CARML or RSL code for the module template. The elements of $\Theta', \Omega', \Pi', \Upsilon'$ have to be contained

in the signature of $\mathcal{C}$ which is given by $\mathfrak{Sig}_{\mathcal{C}} = (\mathfrak{DT}, \mathfrak{Op}, \mathfrak{Pred}, \mathfrak{Var}_{\mathcal{C}})$ where $\mathfrak{DT} = \mathsf{DT} \cup \Theta$, $\mathfrak{Op} = \mathsf{Op} \cup \Omega$, $\mathfrak{Pred} = \mathsf{Pred} \cup \Pi$ and $\mathfrak{Var}_{\mathcal{C}} = \Upsilon$.[1] Optionally, the list of meanings for the uninterpreted symbols in the template for some module $\mathcal{M}$ is followed by a list $(A_1, \dots, A_r; B_1, \dots, B_s)$ of names for the source and sink ports of $\mathcal{M}$. Thus, A_i will serve as name for the i-th source port of the generated instance of $\mathcal{M}$, and B_j as name for the j-th sink port of that instance. The names A_i and B_j have to be pairwise distinct. They can be fresh names or can represent an existing location, in which case consistency of the message-types is required. If A_i is an already existing location then the message-types of A_i and the i-th source port of $\mathcal{M}$ must agree, and from now on location A_i is joined with the i-th source port of the created instance of $\mathcal{M}$. Analogous conditions are required for $B_1, \dots, B_s$ and the sink ports of the created instance of $\mathcal{M}$.

The second type of instantiation is the creation of an entity that is called a *Reo node* [2] which plays a crucial role for the join-operation (explained below). A Reo node can combine zero or more channel ends or I/O-ports of components with the same message-type. Reo nodes can be understood as routers with a special routing strategy. The intuitive meaning is a merger semantics of all read operations performed at the sinks combined in a Reo node N and a replicator semantics of the write operations at the sources of N. That is, all pending read operations at the sinks of N are scheduled in an interleaved way and executed synchronously with writing the received value to all sources of N. The effect of an instantiation of a Reo node via the instruction $\texttt{node}\langle\texttt{type} : T\rangle$ is the creation of a fresh Reo node N without any channel end or I/O-port. The message-type of N is $T \in \mathfrak{DT}$ which indicates that only data items $v \in T$ can flow through N.

Reo operations. RSL supports Reo's main operations for the composition of complex circuits. The join operation $\texttt{join}(N_1, \dots, N_n)$ in RSL takes a list $N_1, \dots, N_n$ of at least two I/O-ports or Reo nodes of the same message-type $T \in \mathfrak{DT}$ as arguments. It creates a new Reo node N of message-type T where all I/O-ports and channel ends of $N_1, \dots, N_n$ are combined. If all N_i are sources then the resulting node N is called source (node). Similarly, N is called sink (node) if all N_i's are sinks. There are more operations offered by Reo, but any circuit which can be constructed by the complete list of Reo operations can also be obtained by a sequence of instantiations and join operations. Thus we omit explanations of additional Reo operations.

Script variables and assignments. I/O-ports, nodes created by a join-operation as well as the result of an instantiation (either a Reo node or module) can be stored into local script variables. Script variables can also be used to hold values of predefined data types (typically an integer value). The script variables are "dynamically typed" and do not have to be declared in advance. An assignment for a script variable sv has the syntax $sv \ := \ \mathcal{V}$ where

[1] There are the obvious side-constraints. If $\texttt{type} : \Theta$ stands for $\texttt{type} : T_1, \dots, \texttt{type} : T_n$ then Θ' must be a list $U_1, \dots, U_n$ of elements in $\mathfrak{DT}$. If $\texttt{op} : \Omega$ encounters m operator symbols then Ω' must be a list of m elements in $\mathfrak{Op}$, and if the i-th element in $\texttt{op} : \Omega$ is $\texttt{op} : (T_{i_1}, \dots, T_{i_k}, T_j)\, f$ then the i-th element of Ω' has to be an element $f' \in \mathfrak{Op}$ of the type $(U_{i_1}, \dots, U_{i_k}, U_j)$. Analogous conditions are required for Π' and Υ'.

$$\mathcal{V} ::= instantiation \mid \texttt{join}(N_1, \ldots, N_n) \mid expression \mid script_variable$$

and *expression* is a term over the signature induced by the predefined data types and the variables $V \in \Upsilon$ (typically arithmetic expressions). Script variables are dynamically sized arrays, with sv being shorthand for $sv[0]$. A script variable sv referring to an instantiated module $\mathcal{M}$ provides access to the interface I/O-ports via $sv.\texttt{source}\,[i]$ and $sv.\texttt{sink}\,[j]$. An RSL circuit can refer to its own source and sink ports via $\texttt{source}[i]$ and $\texttt{sink}[j]$ (see the explanations below).

Control flow instructions. The control flow features (sequential composition, conditional and repetitive commands) have the standard meaning. These and the script variables serve for the stepwise construction of a Reo circuit, and should not be confused with the operational behavior of the network given by the CA for the Reo circuit that results from executing the RSL script. The control flow statements make use of Boolean expressions (bexpr) that impose conditions on the values of script variables and variables in Υ. In the sloppy notation provided for the syntax of for-loops, we assume that i is an integer script variable and j and k are either integer script values or constants.
Interface declaration. In the schema sketched in Fig. 4, the body of an RSL circuit ends with a definition of the nodes that are exported to the higher level as source and sink ports. This can be done in an analogous way as in CARML via "`in`: A_1; ...; `in`: A_r; `out`: B_1; ...; `out`: B_s" to specify that the i-th source port is A_i and the j-th sink port is B_j. It is required that the A_i's are sources and the B_j's are sinks (I/O-ports or Reo nodes) that have been defined in *stmt* via an assignment or module instantiation. Furthermore, the A_i's and B_j's are required to be pairwise distinct. Alternatively, one may depart from the schema in Fig. 4 and define the interface ports in *stmt* via the references $\texttt{source}[i]$ and $\texttt{sink}[j]$, either by assignments ("$\texttt{source}\,[i] := \ldots$" and "$\texttt{sink}\,[j] := \ldots$") or instantiations ("`new` module($\ldots, \texttt{source}[i], \ldots; \ldots, \texttt{sink}[j], \ldots$)"). The indices i and j for the exported source and sink ports have to be consecutive starting with 0. If there are two or more assignments for, e.g., $\texttt{source}\,[i]$ then the last one declares the i-th source port.

Semantics of an RSL circuit. Let $\mathcal{C}$ be an RSL circuit with the parameters Θ, Ω, Π and Υ as in Fig. 4. To provide an operational semantics for $\mathcal{C}$, we fix an interpretation $\mathcal{J}$ for the symbols in the parameter list of $\mathcal{C}$ (which yields an interpretation for the induced signature $\mathfrak{Sig}_{\mathcal{C}}$) and then construct a Reo circuit $\mathcal{R}_{\mathcal{C},\mathcal{J}}$ for $\mathcal{C}$ by means of the instructions given in *stmt*. Finally, we can apply the machinery presented in [5] to construct a constraint automaton $\mathcal{A}_{\mathcal{C},\mathcal{J}}$ from $\mathcal{R}_{\mathcal{C},\mathcal{J}}$. The data-flow vocabulary $\mathfrak{Voc}_{\mathcal{C}}$ of this constraint automaton $\mathcal{A}_{\mathcal{C},\mathcal{J}}$ is defined according to the interface declaration, i.e., $\mathfrak{Voc}_{\mathcal{C}} = (\mathcal{L}_{\mathcal{C}}, \mathcal{L}_{\mathcal{C}}^{src}, \mathcal{L}_{\mathcal{C}}^{snk}, \lambda_{\mathcal{C}})$ where $\mathcal{L}_{\mathcal{C}}^{src} = \{A_1, \ldots, A_r\}$ and $\mathcal{L}_{\mathcal{C}}^{snk} = \{B_1, \ldots, B_s\}$ if the interface declaration specifies $A_1, \ldots, A_s$ as source ports and $B_1, \ldots, B_s$ as sink ports. The function $\lambda_{\mathcal{C}}$ is the obvious one and assigns the message-type of A_i to the i-th source port ($\texttt{source}[i{-}1]$) and the message-type of B_j to the j-th sink port ($\texttt{sink}[j{-}1]$). The set of all observable locations of $\mathcal{C}$ is $\mathcal{L}_{\mathcal{C}} = \mathcal{L}_{\mathcal{C}}^{src} \cup \mathcal{L}_{\mathcal{C}}^{snk}$.

The Reo circuit $\mathcal{R}_{\mathcal{C},\mathcal{J}}$ is obtained by executing the RSL script given by the instructions in the body of $\mathcal{C}$. When instantiating a module the meanings of the uninterpreted data types, operator or predicate symbols are taken according to $\mathcal{J}$. The instantiation of a CARML module $\mathcal{M}(\Theta_0, \Omega_0, \Pi_0, \Upsilon_0)$ with n sources and m sinks via the instruction $comp := \texttt{new}\,\mathcal{M}(\Theta', \Omega', \Pi', \Upsilon')(D_1, \ldots, D_n, E_1, \ldots, E_m)$ means binding an instance *comp* of $\mathcal{M}$ where the i-th source of *comp* is identified with the possibly already existing node D_i and the j-th sink of *comp* with E_j. In the Reo circuit $\mathcal{R}_{\mathcal{C},\mathcal{J}}$, *comp* is viewed as a black-box component. However, by applying the algorithm of [5] (extended to handle global variables) to construct a constraint automaton from $\mathcal{R}_{\mathcal{C},\mathcal{J}}$ we use a constraint automaton $\mathcal{A}_{comp,\mathcal{J}}$ as specification for the behavioral interface of that instance *comp* of $\mathcal{M}$. Automaton $\mathcal{A}_{comp,\mathcal{J}}$ can be obtained as follows. Let $\mathcal{J}_0$ be the interpretation that arises from $\mathcal{J}$ by the substituting Θ_0 with Θ' (i.e., if the i-th data type symbol in Θ_0 is T and the i-th element of Θ' is U then $T^{\mathcal{J}_0} = U^{\mathcal{J}}$) and substituting Ω_0 with Ω', Π_0 with Π', and Υ_0 with Υ'. We now regard the constraint automaton $\mathcal{A}_{\mathcal{M},\mathcal{J}_0}$ and replace the i-th source port of $\mathcal{M}$ with D_i and the j-th sink port of $\mathcal{M}$ with E_j in the data-flow vocabulary of $\mathcal{A}_{\mathcal{M},\mathcal{J}_0}$ and the concurrent I/O-operations that appear as labels for the transitions of $\mathcal{A}_{\mathcal{M},\mathcal{J}_0}$. The resulting automaton is $\mathcal{A}_{comp,\mathcal{J}}$. The meaning of an instantiation of another RSL script $\mathcal{C}'$ via the instruction $comp := \texttt{new}\ \mathcal{C}'(\Theta', \Omega', \Pi', \Upsilon')(D_1, \ldots, D_n, E_1, \ldots, E_m)$ is analogous. It has the effect of including the Reo circuit associated with the generated instance of $\mathcal{C}'$.

```
CIRCUIT buffered_replicator ⟨type : T, var : integer k⟩ {
 // create channels
   F := new FIFO1⟨T⟩(A; R[0]);
   for (i = 1, ..., k){ new SYNC⟨T⟩(R[i]; B[i]); }
 // join channel-ends in a node N
   N := node⟨T⟩;
   for (i = 0, ..., k){ join(R[i], N); }
 // define the interface of the circuit
   source[0] := A;
   for (i = 0, ..., k−1){ sink[i] := B[i + 1]; }
}
```

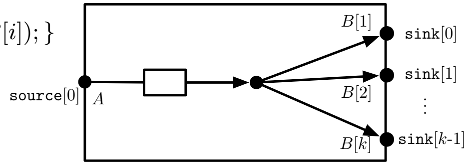


Fig. 5. RSL code for a Reo circuit for a buffered replicator of size k

Example 2. Fig. 5 shows the RSL code and its Reo circuit for a buffered replicator with k output ports where k is an integer variable passed in the parameter list. It uses modules FIFO1⟨type : T⟩ and SYNC⟨type : T⟩ from a built-in library that model a FIFO channel with one buffer cell and a synchronous channel, respectively, where both the input and output port of that channel have (uninterpreted) message-type T. For the instantiation of a buffered replicator circuit one has to provide an interpretation $T^{\mathcal{J}}$ for the data type symbol T to fix the type of data that may flow through the connector and an integer $k^{\mathcal{J}}$ for variable k which

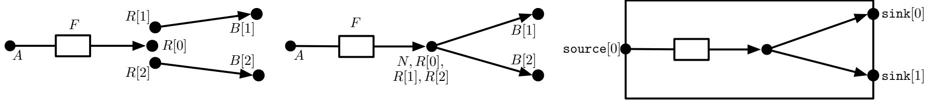


Fig. 6. Three phases for creating a buffered_replicator of size 2. (a) create channels, (b) join channel ends, and (c) export the interface

determines the number of sink ports. (We assume here that $k^{\mathcal{J}} \geq 1$.) Fig. 6 illustrates the three phases in the construction of $\mathcal{R}_{repl,\mathcal{J}}$ (instantiation, composition via the join operations, and interface declaration) for $k^{\mathcal{J}} = 2$. In the first phase of the execution of the RSL script, a FIFO channel with one buffer cell and two synchronous channels are created. In a second phase the source ends of the synchronous channels are joined with the sink of the buffer in a new Reo node N. In the last phase the interface of the connector is defined by exporting the source end of the buffer and the sink ends of the synchronous channels.

```
// include predefined modules and prototype definition for a replicator and module M:
#include "builtin", "buffered_replicator.rsl", "some_component.carml"
CIRCUIT main{
   repl := new buffered_replicator⟨Boolean, 2⟩(C; D1, D2);
   S[0] := new SYNC⟨Boolean⟩(B1; C);
   S[1] := new SYNC⟨Boolean⟩(D1; A1);
   S[2] := new SYNC⟨Boolean⟩(D2; E2);
   comp := new M⟨Boolean⟩(A1, E1; B1);
   in : E1;  out : E2;      // equivalent to source[0] := E1; sink[0] := E2;
}
```

Fig. 7. RSL program for the circuit $\mathcal{R}_{\mathsf{main}}$ depicted in Fig. 8

The replicator can be instantiated in another context like the RSL program in Fig. 7. The instantiation of the buffered replicator yields the interpretation $T^{\mathcal{J}} = \texttt{Boolean}$ and $k^{\mathcal{J}} = 2$. Thus, the first step in the construction of the Reo circuit $\mathcal{R}_{\mathsf{main}}$ for the RSL program in Fig. 7 is running the script for the buffered replicator which yields the (sub)circuit $\mathcal{R}_{repl,\mathcal{J}}$. Then, the three synchronous channels $S[0], S[1]$, and $S[2]$ arise by the instantiation of the CARML module $\mathsf{SYNC}\langle\texttt{type}{:}T\rangle$ taken from the built-in library. Finally, an instance *comp* of a CARML module $\mathcal{M}$ (also taken from a library) is created. We assume here that $\mathcal{M}$'s parameter list consists of an uninterpreted data type symbol and that $\mathcal{M}$ has two input ports and one output port. The first input port of *comp* is identified with node $A1$, the output port of *comp* with $B1$, while the second input port $E1$ of *comp* is a fresh node. Fig. 8 shows the resulting Reo circuit $\mathcal{R}_{\mathsf{main}}$, where *comp* is treated as a black-box component. The constraint automaton $\mathcal{A}_{comp}$ for the behavioral interface of *comp* is obtained from the constraint automaton $\mathcal{A}_{\mathcal{M},\mathcal{J}_0}$ of its prototype module $\mathcal{M}$ and the interpretation $\mathcal{J}_0$ that assigns the type $\texttt{Boolean}$ to the data type symbol T (i.e., $T^{\mathcal{J}_0} = \texttt{Boolean}$) and replacing

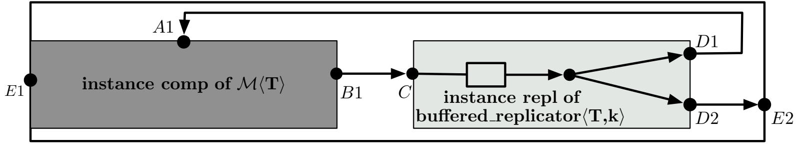


Fig. 8. Reo circuit $\mathcal{R}_{\mathsf{main}}$ for the RSL program in Fig. 7

the names of the I/O-ports of $\mathcal{M}$ by the ones provided during the instantiation. The constraint automaton $\mathcal{A}_{\mathsf{main}}$ for the RSL program in Fig. 7 is then obtained by using the algorithm of [5] to construct a constraint automaton $\mathcal{A}$ for the circuit $\mathcal{R}_{\mathsf{main}}$ (where *comp* is treated as a black-box component), constructing the constraint automaton $\mathcal{A}_{comp} = \mathcal{A}_{\mathcal{M},\mathcal{J}_0}[^{\mathtt{source}[0]}/_{A1}, ^{\mathtt{source}[1]}/_{E1}, ^{\mathtt{sink}[0]}/_{B1}]$, and finally building the product of $\mathcal{A}$ and $\mathcal{A}_{\mathrm{comp}}$ as described in [5]. □

Dynamic reconfiguration. RSL provides support for specifying component connectors with multiple network topologies. The interface of a dynamic connector $\mathcal{C}$ contains a special input port $\mathcal{C}$.`reconf`, called *reconfiguration port*.

```
CIRCUIT C ⟨type : Θ, op : Ω, pred : Π, var : Υ⟩ {
    stmt;                          // construction of a common sub-circuit
    interface_decl             // declaration of the exported source/sink ports
    TOPO(id1) = {stmt1}           // additional sub-circuit for topology id1
      ⋮
    TOPO(idt) = {stmtt}           // additional sub-circuit for topology idt
}
```

Fig. 9. Schema for RSL scripts with dynamic reconfiguration

The schema of RSL scripts with dynamic reconfigurations is shown in Fig. 9. The parameter list is the same as before. The body of a dynamic RSL circuit $\mathcal{C}$ consists of statement *stmt* that specifies a common (static) sub-circuit $\mathcal{C}_{stmt}$ of all network topologies, followed by the interface declaration and instructions of the form $\mathtt{topo}(id_i)\{stmt_i\}$ for $i = 1, \ldots, t$. Here, id_i denotes an identifier for the i-th network topology and $stmt_i$ is a statement. The network topology with an identifier id_i is generated by the composite statement $stmt;\ stmt_i$, resulting in the (static) Reo circuit $\mathcal{C}'_i$. The interface declaration specifies the input and output ports of $\mathcal{C}$, except for the reconfiguration port *circ*.`reconf` of the instances of the circuit $\mathcal{C}$ in Fig. 9. No special declaration is required for the reconfiguration port. The assignments in the interface declaration can only refer to the nodes and ports that appear in *stmt*, but not to entities created in $stmt_1, \ldots, stmt_t$. Any other RSL circuit $circ'$ creating an instance *circ* of the circuit $\mathcal{C}$ in Fig. 9 can access the reconfiguration port *circ*.`reconf` as any port in the interface of *circ*. Hence, any module that is connected in $circ'$ to the reconfiguration port *circ*.`reconf` can serve as a driver and trigger switches in the topology of *circ*

by sending the identifier of the new topology. The constraint automaton $\mathcal{A}_{\mathcal{C},\mathcal{J}}$ for a dynamic connector $\mathcal{C}$ as in Fig. 9 can be seen as a complete hyper-graph with t hyper-vertices (one hyper-vertex for each topology). Each hyper-vertex stands for a constraint automaton for one of the circuits $\mathcal{C}'_i$. The edges of this hyper-graph represent the switch from one topology to another one by receiving a signal on the reconfiguration port. The formal definition of $\mathcal{A}_{\mathcal{C},\mathcal{J}}$ is as follows. The data-flow vocabulary $\mathfrak{Voc}_{\mathcal{C}}$ is defined according to the interface declaration completed with the reconfiguration port $\mathcal{C}.\mathtt{reconf}$ which is a source port of message-type $\{id_1, \ldots, id_t\}$. We use the aforementioned construction for each constraint automaton $\mathcal{A}'_i = \mathcal{A}_{stmt;stmt_i,\mathcal{J}} = (Q^{(i)}, \mathfrak{Voc}_{\mathcal{C}}, \rightarrow_i, Q_0^{(i)})$ for the circuit $\mathcal{C}'_i$ induced by the statement $stmt$; $stmt_i$. The states in $Q^{(i)}$ can be written in the form $\langle q, q^{(i)} \rangle$ where q stands for a state in the constraint automaton $\mathcal{A}_{stmt,\mathcal{J}}$ for the (static) common subcircuit $\mathcal{C}_{stmt}$ of all circuits $\mathcal{C}'_1, \ldots, \mathcal{C}'_t$ and $q^{(i)}$ a state of constraint automaton $\mathcal{A}_{stmt_i,\mathcal{J}}$ for the subcircuit induced by $stmt_i$. W.l.o.g. we can assume that $Q^{(i)} \cap Q^{(j)} = \emptyset$ for $1 \leq i < j \leq t$. The constraint automaton $\mathcal{A}_{\mathcal{C},\mathcal{J}} = (Q, \mathfrak{Voc}_{\mathcal{C}}, \rightarrow, Q_0)$ for the dynamic connector $\mathcal{C}$ is then obtained by combining $\mathcal{A}'_1, \ldots, \mathcal{A}'_t$ as follows. The state space Q of $\mathcal{A}_{\mathcal{C},\mathcal{J}}$ is the disjoint union of the state spaces of $\mathcal{A}'_1, \ldots, \mathcal{A}'_t$, that is, $Q = Q^{(1)} \cup \ldots \cup Q^{(t)}$. The set Q_0 of initial states in $\mathcal{A}_{\mathcal{C},\mathcal{J}}$ is the set of all states $\langle q, q^{(i)} \rangle$ where q is an initial state in the constraint automaton $\mathcal{A}_{stmt,\mathcal{J}}$ for $stmt$ and $q^{(i)}$ an initial state in the constraint automaton $\mathcal{A}_{stmt_i,\mathcal{J}}$ for $stmt_i$. The transitions $\rightarrow$ of $\mathcal{A}_{\mathcal{C},\mathcal{J}}$ are given by the following two rules, where the first stands for the receipt of the signal to switch to the j-th network topology and the second rule stands for the execution of a concurrent I/O-operation in the i-th topology:

$$\frac{q^{(j)} \text{ initial state of the CA } \mathcal{A}_{stmt_j,\mathcal{J}} \text{ for } stmt_j}{\langle q, q^{(i)} \rangle \xrightarrow{\mathcal{C}.\mathtt{reconf}?id_j} \langle q, q^{(j)} \rangle} \qquad \frac{\langle q, q^{(i)} \rangle \xrightarrow{c}_i \langle q, p^{(i)} \rangle}{\langle q, q^{(i)} \rangle \xrightarrow{c} \langle q, p^{(i)} \rangle}$$

where $\mathcal{C}.\mathtt{reconf}?id_j$ denotes the unique concurrent I/O-operation c with $\mathcal{L}(c) = \{\mathcal{C}.\mathtt{reconf}\}$ and $c(\mathcal{C}.\mathtt{reconf}) = id_j$.

5 Modeling a Railway Network

We demonstrate our hybrid approach with Vereofy's input languages CARML and RSL by means of a toy example modeling a simple railway network, composed out of basic building blocks (tracks, stations, switches). Trains are represented by unique identifiers and travel of a train is modeled by data flow of its identifier. We use the CARML module track (Fig. 3) as the basic (unidirectional) railway track, allowing a train to stop or to pass through instantaneously. A CARML module track_with_train is a variant of this basic track, initially occupied by a train. It has an additional parameter (var: *TrainType initial_train*) used to specify the identifier of this train and can be derived from track by setting the initial values of variables "stat := occupied" and "train := initial_train". A CARML module for a train station (train_station) is obtained from track by removing the transition for an instantaneous pass-through.

Railway switches come in two variants, left-hand-side (lhs) with k entries and one exit and right-hand-side (rhs) with k exits and one entry. In Fig. 10, we show how to model two variants of right-hand-side switches with nondeterministic choice between the possible exits, one as a CARML module with two exits (simple_rhs_switch) and one as a RSL script with a parameterized number of k exits (rhs_switch), recursively built out of simpler switches. As an example for a left-hand-side switch, Fig. 10 also shows the RSL script for a switch with two entries, using dynamic reconfiguration (reconf_lhs_switch).

The CARML module for simple_rhs_switch ensures that a train leaves via exactly one exit and serves as the basic building block for the more general rhs_switch. Its RSL script has a parameter k specifying the number of exits (sink ports). The first two lines handle the base cases, by instantiating either a built-in synchronous channel or a simple_rhs_switch and exporting their ports. For $k > 2$, two instances of rhs_switch with half the number of exits are recursively instantiated (r_1,r_2) and a simple_rhs_switch (l) switches between r_1 and r_2. In the last three lines, the interface with one source port (that of l) and k sink ports (those of r_1 and r_2) is generated.

The RSL script for reconf_lhs_switch declares two sources and one sink as the common interface. In both topologies (having identifiers 0 and 1), one of the sources is connected via a synchronous channel to the sink, the other source is left unconnected (and thus blocks data flow). Upon receipt of topology identifier i at the reconfiguration port, the switch reconfigures to topology i, letting trains pass only from source i to the sink port.

The RSL program in Fig. 11 composes a simple railway network. A set of train identifiers is defined and the building blocks are included. In the RSL script main, instances of the building blocks are created and connected at the nodes L_i to yield the depicted network. To provide reconfiguration signals, an instance of the CARML module driver is connected to the reconfiguration port of sw_1, alternately sending the topology identifiers 0 and 1.

The given model of the railway example may now serve as input for our verification toolkit Vereofy. The tool allows to check safety or liveness conditions specified by temporal formula with classical modalities, but also to argue about the observable data flow at the locations of the network [4].

6 Implementation

Our toolkit Vereofy (see Fig. 1 and `www.vereofy.de`) supports modeling using RSL and CARML. Vereofy can be used as stand-alone tool or as an Eclipse plugin for the Eclipse Coordination Tools (ECT) [19] which allow to specify connectors in a graphical way. It currently supports all language features explained in Sections 3 and 4, except for the parameterization by uninterpreted symbols for data types, operators and predicates. Furthermore, global variables and locations of different message-types are not yet supported[2]. For model checking purposes,

[2] In Sections 3 and 4 we focused on a clear presentation of the core features of CARML and RSL and slightly departed from the syntax used in our implementation.

```
MODULE simple_rhs_switch⟨type : TrainType⟩ {
   in :  TrainType A;
   out : TrainType B;  out : TrainType C;
   −[ active(A) ∧   active(B) ∧ ¬active(C) ∧ data_A = data_B ]→;
   −[ active(A) ∧ ¬active(B) ∧   active(C) ∧ data_A = data_C ]→;
}  CIRCUIT rhs_switch⟨type : TrainType,  var : integer k⟩ {
   if(k = 1){new SYNC⟨TrainType⟩(source[0]; sink[0]); }
   if(k = 2){new simple_rhs_switch⟨TrainType⟩(source[0]; sink[0], sink[1]); }
   if(k > 2){
     l := new simple_rhs_switch⟨TrainType⟩;
     r1 := new rhs_switch⟨TrainType, ⌈k/2⌉⟩;
     r2 := new rhs_switch⟨TrainType, k − ⌈k/2⌉⟩;
     join(l.sink[0], r1.source[0]); join(l.sink[1], r2.source[0]);
     source[0] := l.source[0]; out := 0;
     for(i = 1, . . . , ⌈k/2⌉) {sink[out] := r1.sink[i − 1]; out := out + 1; }
     for(j = 1, . . . , k − ⌈k/2⌉) {sink[out] := r2.sink[j − 1]; out := out + 1; }
   }
}  CIRCUIT reconf_lhs_switch⟨type : TrainType⟩ {
   source[0] := node⟨TrainType⟩; source[1] := node⟨TrainType⟩;
   sink[0] := node⟨TrainType⟩;
   TOPO(0) = {new SYNC⟨TrainType⟩(source[0]; sink[0]); }
   TOPO(1) = {new SYNC⟨TrainType⟩(source[1]; sink[0]); }
}
```

Fig. 10. Specifications for three variants of railway switches

```
TYPE Trains = enum{T1, T2, no_train};
#include "builtin","railway_building_blocks"
MODULE driver {
   out : int(0, 1) B;    var : int(0, 1) s := 0;
    −[ active(B) ∧ data_B = s ]→ s := (s + 1) mod 2;
}
CIRCUIT main {
   t1 := new track_with_train⟨Trains, no_train, T1⟩(L6; L1);
   t2 := new track_with_train⟨Trains, no_train, T2⟩(L5; L3);
   t3 := new track⟨Trains, no_train⟩(L2; L4);
   st := new station⟨Trains, no_train⟩(L4; L7);
   sw1 := new reconf_lhs_switch⟨Trains⟩;
   sw2 := new rhs_switch⟨Trains, 2⟩;
   join(L1, sw1.source[0]); join(L3, sw1.source[1]);
   join(L2, sw1.sink[0]);
   join(L5, sw2.sink[0]); join(L6, sw2.sink[1]);
   join(L7, sw2.source[0]);
   d := new driver; Lr = join(d.sink[0], sw1.reconf);

}
```

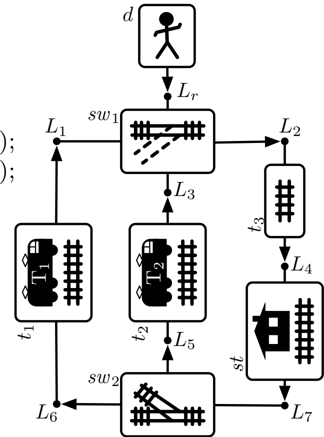


Fig. 11. Example RSL program for a railway network with two trains

RSL programs are translated into a symbolic BDD-based representation as presented in [6] of the corresponding constraint automaton. The translation is done by constructing the Reo circuit and applying the machinery presented in [5] and the enhancements for dynamic connectors explained in Section 4.

number of phils	**reachable states**	**BDD nodes**	**building time (s)**
100	$1,89 \cdot 10^{38}$	15618	7, 29
200	$3,59 \cdot 10^{76}$	31418	31, 15
400	$1,29 \cdot 10^{153}$	63018	148, 87
600	$4,62 \cdot 10^{229}$	94618	364, 35
800	$1,66 \cdot 10^{306}$	126218	706, 12
1000	$> 10^{308}$	157818	1157, 25

Fig. 12. Dining philosophers results

number of processes	**reachable states**	**BDD nodes**	**building time (s)**	**reachable time (s)**
(50, 29)	$2,54 \cdot 10^{21}$	2652	8, 76	0, 46
(50, 50)	$7,54 \cdot 10^{28}$	5298	10, 13	0, 95
(100, 61)	$1,22 \cdot 10^{45}$	8412	160, 59	4, 72
(100, 100)	$6,78 \cdot 10^{58}$	18123	192, 61	13, 25
(120, 79)	$5,00 \cdot 10^{56}$	12807	743, 17	12, 57
(120, 120)	$6,81 \cdot 10^{70}$	25353	785, 89	164, 97

Fig. 13. Mutual exclusion results

We illustrate the scalability of our approach by two examples. The first is a variant of a dining philosophers example [14]. The table in Fig. 12 shows the time in seconds needed to synthesize the scenario for a given number of philosophers, the number of reachable states as well as the number of BDD nodes necessary to store the composite system. Computing the reachable fragment of the state space finishes within one second for all depicted sizes. The second example is a mutual exclusion protocol using exogenous coordination, where n processes are present and k are allowed to enter their critical section at the same time. The table in Fig. 13 shows the time for the synthesis of the system and the time for computing the reachable fragment of the state space for different values of (n, k). The results for both examples have been achieved on a 2,2GHz CPU and 2GB memory.

7 Related Work and Conclusion

Related work. For the design of our languages RSL and CARML we borrowed ideas from many other modeling and coordination languages. We argue that there are rather natural transformations of many other languages into our hybrid modeling approach. Several formalisms have been embedded into Reo, such as Petri nets [20], the actor-based language Rebeca [21] or UML sequence diagrams [3], which can immediately encoded in RSL. The main features of process algebras can be mimicked by Reo operations and encoded in RSL. E.g., CCS-like parallel composition with synchronization over complementary actions can be modeled by synchronous channels and the join-operation. Nondeterministic choice operators can be modeled by a (RSL script for a) Reo circuit for an exrouter [2]. The concept of name-passing as in the π-calculus [18] is not yet supported by RSL, but intended for a future extension of RSL.

Interaction systems were introduced in [12] as a general model for component-based systems. In this approach, the behavioral interfaces of components are modeled by labeled transition systems and they offer ports to communicate

with each other. Up to some syntactic adaptions for communication actions, they can easily be specified in CARML. Connectors of an interaction system are used to glue ports of different components together by enforcing some actions to be synchronized. They have a natural representation by an RSL script which instantiates several synchronous channels and performs several join operations on their channel ends.

Although the syntax of CARML is inspired by reactive modules [1] there are some crucial differences concerning the interactions of modules. Communication of reactive modules has to be realized via interface variables and the parallel composition of reactive modules is defined in terms of rounds. This round-based coordination principle of reactive modules can be modeled by a Reo circuit specified by an RSL script.

Conclusion. The presented approach is based on two modeling languages RSL and CARML which together permit formal reasoning about component-based systems relying on endogenous and exogenous coordination, possibly with dynamic reconfigurations of the network topology. It allows for compositional and hierarchical design and reusability of components and coordination units. In our opinion, our hybrid approach yields a good compromise between (1) the elegance and expressiveness of coordination languages and (2) meta-languages supporting an efficient generation of a compact system-representations that yield the basis for applying model checking routines.

To illustrate the main features of our hybrid approach, we presented a toy example and experimental results for the model generation with Vereofy for academic case studies. We are currently working on the modeling and verification of larger examples with our tool set, such as a peer-to-peer protocol with a dynamic network manager and a bio-medical sensor network. The embeddings of other languages as sketched in Section 7 together with the wide range of application areas of Reo (such as modeling of compliance-aware business processes, long-run business transactions, and orchestration of web services [19]) makes our modeling and verification approach with the tool set Vereofy applicable for many purposes.

References

1. Alur, R., Henzinger, T.: Reactive Modules. Formal Methods in System Design: An Intern. J. 15(1), 7–48 (1999)
2. Arbab, F.: Reo: A Channel-Based Coordination Model for Component Composition. MSCS 14(3), 329–366 (2004)
3. Arbab, F., Sun, M., Baier, C.: Synthesis of Reo circuits from scenario-based specifications. In: FOCLASA 2008. ENTCS (2008) (to appear)
4. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: Formal verification for components and connectors (submitted for publication)
5. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling Component Connectors in Reo by Constraint Automata. SCP 61, 75–113 (2006)
6. Blechmann, T., Baier, C.: Checking equivalence for reo networks. In: FACS 2007. ENTCS, vol. 215, pp. 209–226 (2008)

7. Blechmann, T., Klein, J., Klüppelholz, S.: Vereofy, http://www.vereofy.de/
8. Capizzi, S., Solmi, R., Zavattaro, G.: From endogenous to exogenous coordination using aspect-oriented programming. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949. Springer, Heidelberg (2004)
9. de Alfaro, L., Dias da Silva, L., Faella, M., Legay, A., Roy, P., Sorea, M.: Sociable interfaces. In: Gramlich, B. (ed.) FroCos 2005. LNCS, vol. 3717, pp. 81–105. Springer, Heidelberg (2005)
10. de Alfaro, L., Henzinger, T.: Interface Theories for Component-Based Design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)
11. Gelernter, D., Carriero, N., Chandran, S., Chang, S.: Parallel programming in linda. In: ICPP, pp. 255–263 (1985)
12. Gößler, G., Sifakis, J.: Component-based construction of deadlock-free systems: Extended abstract. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 420–433. Springer, Heidelberg (2003)
13. Guillen-Scholten, J., Arbab, F., de Boer, F., Bonsangue, M.: MoCha-pi: an exogenous coordination calculus based on mobile channels. In: Proceedings of the 2005 ACM symposium on Applied computing (SAC), pp. 436–442. ACM, New York (2005)
14. Klüppelholz, S., Baier, C.: Symbolic Model Checking for Channel-based Component Connectors. In: Science of Computer Programming (2009)
15. Klüppelholz, S., Baier, C.: Alternating-time stream logic for multi-agent systems. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 184–198. Springer, Heidelberg (2008)
16. Lynch, N., Tuttle, M.: An Introduction to Input/Output Automata. CWI Quarterly 2(3), 219–246 (1989)
17. Majster-Cederbaum, M., Minnameier, C.: Everything is PSPACE-complete in interaction systems. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 216–227. Springer, Heidelberg (2008)
18. Milner, R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, Cambridge (1999)
19. Reo website at CWI Amsterdam, http://reo.project.cwi.nl/
20. Scholten, J.-G., Arbab, F., de Boer, F., Bonsangue, M.: Modeling the exogenous coordination of mobile channel-based systems with Petri nets. In: FOCLASA 2005. ENTCS, vol. 154(1), pp. 83–99 (2006)
21. Sirjani, M., Jaghoori, M., Baier, C., Arbab, F.: Compositional semantics of an actor-based language using constraint automata. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 281–297. Springer, Heidelberg (2006)