Parallel Constraint Automata

Benjamin Lion[1] and Farhad Arbab[2,3]

[1]INRIA, Rennes
[2]Computer Security, CWI, Amsterdam
[3]LIACS, Leiden University

Abstract

In this paper, we introduce Parallel Constraint Automata (PCA), a variant of the original Constraint Automata (CA) better suited to compactly represent highly concurrent compositional models, such as Reo. By relegating the crux of composition to run-time, the PCA models maintain the integrity of the constituent parts of a composite model, allowing dynamic composition and modification of PCA models. We discuss the run-time structure of the PCA with their implementation in Maude, and the notion of maximality for a composite transition with respect to various criteria. We show the utility of PCA in formal analysis through Maude search queries.

1 Introduction

Reo [2] is an exogeneous coordination language in which interaction protocols are built compositionally out of a set of (user defined) primitive channels. In Reo, a channel denotes a constraint on the flow of data through the two ports at its boundary, and as such it represents a binary interaction protocol. Composing such channels (i.e., binary protocols) yields more complex interaction protocols which accord an intuitive graphical representation as connectror circuits. A small set of very simple primitive channels suffices to compose arbitrarily complex protocols. Over the years, several semantics have been defined to express the dataflow nature of primitive channels, and how primitives compose to create complex circuits.

Constraint Automata (CA) [7] were introduced as an operational semantics for Reo, primarily intended to express the semantics of Reo circuits for the purpose of formal verification through model checking [6, 20]. Subsequently, variants of CA and other automata-based models inspired by CA were also used as operational models based on which Reo compilers translate Reo circuits into executable code [17, 16, 15], to model quality of service [26, 30, 3], as well as in the synthesis of Reo circuits from CA [27] and from specification of concurrency protocols expressed in other models, such as UML sequence

diagrams [5] and BPMN business process models [4]. However, like most (but not all) formal semantics of Reo [14], the original CA is an inherently sequential model that captures the highly concurrent nature of Reo only indirectly through explicit enumeration of all valid combinations of states and transitions of the constituents that comprise a composite model. Such enumeration results in a combinatorial explosion that becomes an obstacle, especially when CA are to be used as an operational model for simulation or execution.

Moreover, the composition operation on CAs loses the structure of the composite CAs by merging states and transitions. The lack of means to isolate individual CAs after composition limits the kinds of structural runtime analysis that can be performed. Indeed, the ability to isolate a specific CA out of a composition can lead to insightful understanding of how a process behaves within its changing environment (i.e., the other CAs in that or another composition).

The original constraint automata model was meant to capture the semantics of Reo circuits for static verification. In this paper, we present Parallel Constraint Automata (PCA), a variant of the original CA meant for execution, dynamic reconfiguration, and dynamic verification. As an executable, verifiable, and reconfigurable model, the PCA must satisfy the following list of properties:

1. soundness: every trace produced at runtime is sound with respect to the specification;

2. completeness: every trace possible in the specification is reachable at runtime;

3. realistic: a step from runtime composition succeeds only if it reflects a permissible dataflow;

4. structure preserving: the runtime structure reflects the (modular) structure of the specification;

5. expressive: the runtime is sufficiently expressive to contextualize the execution (e.g., optimize for throughput, for fairness, for maximal progress);

6. verifiable: the executable model is formal and can support verification.

We evaluate the related work from the perspective of the above list of objectives, and show how our new model adequately addresses all of them.

Existing compilers for Reo either use a static composition (as in the Treo compiler [8, 13]), or some form of hybrid composition (as in the Lykos compiler [16]) that partitions the set of constraint automata into synchronous subsets. The first approach is not structure preserving, as the composition is done statically. Similarly, although less restrictive, the second approach assumes static composition of synchronous subsets, which is also not structure preserving. Moreover, the runtime composition of synchronous subsets is not associative, i.e., CAs at runtime must execute in a specific order for the runtime to avoid deadlock. In our case, the runtime allows arbitrary multisets of CAs to run in parallel (in Section 2.2.2), which preserves the full structure of the

original specification expression. Our underlying composition operator forms a multiset of connector and is therefore commutative and associative, which is closer to the original composition operator in Reo. Our model supports equational reasoning, e.g., in Maude, to equationally rewrite a synchronous subset to a simpler component (as we show in Section 3.2).

The Dreams framework [31] is an approach that composes CAs at runtime. In the Dreams framework, coordination is achieved by a consensus among a distributed set of CAs. In the Dreams framework, ports are not explicit: they are attributes of actors. We chose to make ports explicit structural constructs in our framework, instead of attributes of CAs. Consequently, there is no need in our model to run a centralized consensus algorithm among CAs, as one failure detected at a port triggers an error whose propagation prunes wrong choices (see Section 2.2.2). To achieve the goal of providing a realistic runtime, our execution engine makes sure that transitions in CAs run in the order of the dataflow through ports.

The constraint satisfaction approach [10, 9] takes a different perspective on the problem of coordination: every constraint is a logical formula, and a centralized solver provides a sequence of consistent solutions to those constraints over time. The benefit of such approach is that its logic is expressive enough to encode higher order constraints such as context dependent constraints. However, not every solution found declaratively by a solver necessarily reflects a realistic dataflow through a Reo connector, as it may violate causality. We show an algorithm that solves the assignment of data to ports while still preserving causality (i.e., a port gets a value before being used) that reflects the realistic nature of dataflow through Reo connectors.

In addition to the points already mentioned above, our runtime provides two new features. First, our use of rewriting logic opens new avenues for verification of the correctness of connectors. Our Maude runtime provides a syntax to perform reachability queries on a composition of connectors. This way, one can search (see Section 3.2) to verify if a CA state is reachable from an initial state, within a context (i.e., a composition of CAs).

Second, the structure and modularity of our runtime opens possibilities to express new kinds of components. For instance, we can model components that reconfigure a component with a rewriting rule that triggers its reconfiguration in a specific state. Moreover, in our rewriting logic framework we can combine CAs with non-CA models in heterogeneous specifications.

The structure preserving feature of our executable environment forces our product to be lazy, at runtime. Therefore, the Maude implementation provides a "just in time" composition of the constraint automata that is sound with respect to the static composition in existing work. A similar strategy is used in tools such as Uppaal [22]. Uppaal checks a temporal formula in CTL using a network of timed automata. The underlying calculus of networks of timed automata uses a quotient formula and constraint solving methods. In our Maude implementation, verification is done via reachibility queries on the composition. The language for reachability queries is quite liberal, and can be any pattern of the system's description. Instead of asking, as in Uppaal "is this formula true

within the context of the composition $A_1 \mid \cdots \mid A_n$?", we ask "is the pattern X reachable after execution of the composition $A_1 \mid \cdots \mid A_n$?". We leave as future work the development of a dedicated syntax for useful search queries (and a calculus similar to that in Uppaal to further simplify the search). See [1] for a work to relate the verification of CTL formulas and reachibility queries on a network of automata.

2 Parallel Constraint Automata

In this section, we give a brief introduction to constraint automata as first presented in [7], and then extended with memory in [18]. Constraint automata have been applied in the fields of design and verification of networks of interacting components [7, 19, 20].

We start with few prerequisites, introduce our notation, and then present in Section 2.1 the syntax and semantics for constraint automata. We give two properties that are generally assumed: the fact that the constraint labeling a transition restricts the data flowing at all ports of a constraint automaton (i.e., there is no data flow through a port that does not appear in the transition); and that a silent transition is implicit in every state. We say that a constraint automaton is *guard covered* if it satisfies the first property and *independent* if it satisfies the second.

Next, in Section 2.2, we present an algorithm to compose constraint automata in parallel. The algorithm ensures that the assignment of values to ports is sound with respect to the static product of constraint automata.

Prerequisites We use $\mathbb{D}$ to denote the set of all data elements, and $\mathcal{N}$ the set of all variable names. In our context, a variable name can be either a port name (regardless of its input/output directionality), or a memory name.

We write an assignment as a function of signature $\mathcal{N} \mapsto \mathbb{D}$. Conceptually, an assignment is a collection of simultaneous events that occur at the ports and memories in $\mathcal{N}$.

A stream of assignments is an element of $(\mathcal{N} \to \mathbb{D})^\omega$, which denotes an infinite sequence of assignments. For $\sigma : X \to Y$, we use $dom(\sigma)$ to refer to the domain of σ, i.e., the set X.

2.1 Constraint automata

Reo connectors synchronize through shared ports. The standard operational specification of a Reo connector is defined as a constraint automaton. In this section, we give a brief introduction to constraint automata with memory; for more precise description, see [18].

A constraint automaton has labeled transitions among its states. The label of a transition is a formula that we call a *guard*. A guard consists of a set of free memory and port variables, and a constraint that restricts the data assignments to those variables. The state based description of a constraint

automaton, therefore, encodes sequences of solutions to interaction constraints, formally modeled as solutions to the guards labeling its transitions.

Guards (syntax) For $* \notin \mathbb{D}$, we use $\mathbb{D}_\star = \mathbb{D} \cup \{\star\}$, and use `Some` d to denote an element $d \in \mathbb{D}$. We use $\mathcal{N} = \mathcal{N}_P \cup \mathcal{N}_L \cup \mathcal{N}_M \cup \mathcal{N}_{M^\bullet}$ as a set of variables, where $\mathcal{N}_P, \mathcal{N}_L$, and $\mathcal{N}_M$ are mutually disjoint. A term, typically denoted as $t_1, t_2, ...$, is an an element of $\mathcal{N} \cup \mathbb{D}_\star$. We syntactically differentiate between *port* names, typically denoted as $p_1, p_2, ... \in \mathcal{N}_P$, *data variables* names, typically denoted as $x_1, x_2, ... \in \mathcal{N}_L$, *current memory* names, denoted as $m_1, m_2, ... \in \mathcal{N}_M$, and *next memory* names, which we denote as $m_1^\bullet, m_2^\bullet, ... \in \mathcal{N}_{M^\bullet}$. We assume that the two sets $\mathcal{N}_M$ and $\mathcal{N}_{M^\bullet}$ are in a bijection, i.e., every memory name $m \in \mathcal{N}_M$ has a next memory name $m^\bullet \in \mathcal{N}_{M^\bullet}$ and vice versa. Note that, for simplicity, we do not consider function symbols in the set of terms.

The set of *guards* G is the set of all equality of terms $t_1 = t_2$ closed under the conjunction operator, i.e., given $g_1 \in G$ and $g_2 \in G$, then $g_1 \wedge g_2 \in G$. Note that, for simplicity, we do not consider predicate symbols. We assume $\mathbb{D}$ to contain at least one data element d, and we write as a shorthand notation $\top := \star = \star$ and $\bot := \star =$ `Some` d. We use $\mathcal{N}(g)$ to denote the set of free variable names in $g \in G$. Given a guard g such that $v \in \mathcal{N}(g)$, we write $g[v \mapsto d]$ for the new formula where the free variable v is substituted by a constant $d \in \mathbb{D}_\star$.

Similarly, an empty memory cell is encoded with the constraint $m = \star$. Finally, we denote the firing of a port as $p =$ `Some` x, where x is a data variable in $\mathcal{N}_L$.

Guards (semantics) We use the standard satisfaction relation between guards and assignments. Port and memory variables are interpreted in the domain $\mathbb{D}_\star$, i.e., can be assigned the $\star$ value, and data variables are interpreted in the domain $\mathbb{D}$, i.e., are always assigned to `Some` d for some $d \in \mathbb{D}$.

Given a guard $g \in G$, we say that $\Gamma \in \mathcal{N} \to \mathbb{D}_\star$ satisfies g, and write $\Gamma \models g$, defined inductively on g as:

- $\Gamma \models t_1 = t_2$ if and only if $\Gamma(t_1) = \Gamma(t_2)$ (where $\Gamma(t) = t$ if t is a constant symbol in $\mathbb{D}_\star$); and
- $\Gamma \models g_1 \wedge g_2$ if and only if $\Gamma \models g_1$ and $\Gamma \models g_2$.

Constraint automata (syntax) A *constraint automaton* is a tuple $A = (Q, \mathcal{V}, \to, Q_0)$, where Q is a set of *state* names, $\mathcal{V} \subseteq \mathcal{N}$ is a set of variable names, $\to \in Q \times G \times Q$ is a set of *transitions* labeled with a guard $g \in G$ such that $\mathcal{N}(g) \subseteq \mathcal{V}$, and $Q_0 \in 2^Q$ is a set of *initial* states. We use the shorthand notation $q \xrightarrow{g} q'$ for transitions $(q, g, q') \in \to$.

Note that in the standard definition of constraint automata, a transition is also labeled with the set $\mathcal{N}(g) \cap \mathcal{N}_P$ of free port variables in the guard, called the port synchronization set. As this set can be recovered, we drop it from transition labels in our notation.

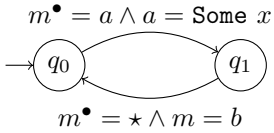

Figure 1: `Fifo(a,b)`

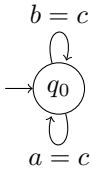

Figure 2: `Merger(a,b,c)`

In Figure 1 and Figure 2, we show two constraint automata for the Fifo and the Merger, respectively. Note that the syntax `Some` x enforces the value assigned to a to be an element different from $\star$. We show in Table 1 and Table 2 a prefix for an execution of, respectively, the Fifo and the Merger constraint automata.

Constraint automata (semantics) The semantics of a constraint automaton captures the series of data flowing at ports and memory locations of that constraint automaton. In the case where there is no data flow, we use the constraint $p = \star$ to denote that p is assigned the silent element $\star$.

Let $A = (Q, \mathcal{V}, \rightarrow, Q_0)$ be a constraint automaton. A *run* $\sigma \in \rightarrow^\omega$ of the automaton A, is an infinite sequence of consecutive transitions. We refer to the i^{th} element of σ as $\sigma_i = q_i \xrightarrow{g_i} q_{i+1}$ and we impose that $\sigma_0 = q_0 \xrightarrow{g_0} q_1$ with $q_0 \in Q_0$. An *execution* of a run σ is a stream of observations $\gamma \in (\mathcal{V} \rightarrow \mathbb{D})^\omega$ such that for all $i \in \mathbb{N}$, $\gamma(i) \models g_i$, and for all $m \in \mathcal{V} \cap \mathcal{N}_M$, $\gamma(i{+}1)(m) = \gamma(i)(m^\bullet)$. The second condition semantically relates the two memory variables $m \in \mathcal{V} \cap \mathcal{N}_M$ and $m^\bullet \in \mathcal{V} \cap \mathcal{N}_{M^\bullet}$ as, respectively, the current and the next values of the same memory variable m. If a guard g_i does not constrain a port variable $p \in \mathcal{V} \cap \mathcal{N}_P$, we semantically assume that the variable is restricted to $\star$, i.e., that the port p does not fire. If a guard g_i does not constrain a next memory variable $m^\bullet \in \mathcal{V} \cap \mathcal{N}_{M^\bullet}$, we semantically assume $m = m^\bullet$, i.e., that the data value of the memory is propagated to the next step of the execution. If the run contains a guard that evaluates to false, then no execution exists for this run.

The *behavior* $L(A)$ of a constraint automaton A is the set consisting of all executions of all runs on A, i.e., a subset of $(\mathcal{V} \rightarrow \mathbb{D})^\omega$. Note that the standard semantics of constraint automata restricts the set $\mathcal{V}$ to the set of port names only, whereas our semantics also deals with memory variables. As this difference is not the main motivation in this paper, we ignore such detail.

a	m	b
d_1	d_1	$\star$
$\star$	d_1	$\star$
$\star$	d_1	d_1
$\cdots$	$\cdots$	$\cdots$

Table 1: An execution for `Fifo(a,b)`.

a	b	c
d_1	$\star$	d_1
$\star$	d_2	d_2
$\star$	$\star$	$\star$
$\cdots$	$\cdots$	$\cdots$

Table 2: Execution for `Merger(a,b,c)`.

Composition Given two constraint automata $A_i = (Q_i, \mathcal{V}_i, \rightarrow_i, Q_{0,i}), i \in \{1,2\}$ such that $(\mathcal{V}_1 \setminus \mathcal{N}_P) \cap (\mathcal{V}_2 \setminus \mathcal{N}_P) = \emptyset$ (i.e., only port variables may be shared), we call $A = (Q_1 \times Q_2, \mathcal{V}_1 \cup \mathcal{V}_2, \rightarrow, Q_{0,1} \times Q_{0,2})$ the composition of A_1 and A_2, and write $A = A_1 \times A_2$ where:

$$\frac{q_1 \xrightarrow{g_1} q_1' \qquad q_2 \xrightarrow{g_2} q_2'}{(q_1, q_2) \xrightarrow{g_1 \wedge g_2} (q_1', q_2')} \tag{1}$$

The behavior of a composite constraint automaton $A_1 \times A_2$ is equal to the merging of the behaviors of its operands A_1 and A_2 (see a similar proof in [7]), i.e., $L(A_1 \times A_2) = L(A_1) \bowtie L(A_2)$, where $\sigma \in L(A_1) \bowtie L(A_2)$ if and only if there exists $(\sigma_1, \sigma_2) \in L(A_1) \times L(A_2)$, such that for all $i \in \mathbb{N}$, $k \in \{1,2\}$, and $v \in \mathcal{V}_k$, $\sigma_k(i)(v) = \sigma(i)(v)$. When $\mathcal{V}_1 = \mathcal{V}_2$, the product leads to the intersection of executions, i.e. $L(A_1) \bowtie L(A_2) = L(A_1) \cap L(A_2)$.

Properties Recall that we semantically enforce that every port variable of a constraint automaton that does not appear in a guard is assigned $\star$. We can also syntactically enforce this property by considering guards that cover all variables. A guard g covers a set of variables V if every variable in the set V appears free in g.

Theorem 1 (Guard-covered)**.** *A constraint automaton $A = (Q, \mathcal{V}, \rightarrow, Q_0)$ is semantically equivalent to another constraint automaton, called guard-covered, for which all guards cover $\mathcal{V}$. The guard-covered property is preserved through composition.*

Proof. If a variable in $\mathcal{V}$ does not appear free in g, the satisfaction relation that is accepted by the semantics of the constraint automaton either imposes that $p \mapsto \star$ if the variable is a port variable, or $m = m^\bullet$ for a memory variable. Adding those clauses as conjuncts to g does not change the semantics, and makes g cover $\mathcal{V}$. □

Figures 3 and 4 show the two guard-covered constraint automata for the Fifo and the Merger, respectively.

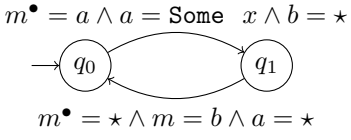

Figure 3: `Fifo(a,b)`

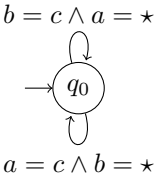

Figure 4: `Merger(a,b,c)`

The standard definition of constraint automata in [7] has three rules for composition: the two other rules encode the *independent* transitions of one constraint automaton while the other one stays in the same state. Instead, we

eliminate the two independence rules and impose a syntactic constraint on constraint automata to recover independent transitions. A constraint automaton is *independent* if every one of its states has a self-loop transition with the guard $\top$. Consequently, the behavior of an independent automaton is closed under insertion of arbitrary silent steps for ports, while it preserves its memory values across a run.

Theorem 2 (Independence)**.** *A constraint automata $A = (Q, \mathcal{V}, \rightarrow, Q_0)$ for which the transition relation contains, for all $q \in Q$, the transition $q \xrightarrow{\top} q$ is closed under insertion of arbitrarily many (possibly infinite) assignments of silent value to ports, while preserving memory values. The independence property is preserved through composition.*

Proof. The guard $\top$ is semantically equivalent to the conjunction of a clause $p = \star$ for every $p \in \mathcal{V} \cap \mathcal{N}_P$ and a clause $m^\bullet = m$ for every $m \in \mathcal{V} \cap \mathcal{N}_M$. Thus, an execution can be extended with arbitrarily many assignments that satisfy the new $\top$ constraints, from any state. Because the conjunction of two $\top$ constraints is still $\top$, the independence is preserved through composition. □

Figures 5 and 6 show the two independent constraint automata for the Fifo and the Merger, respectively.

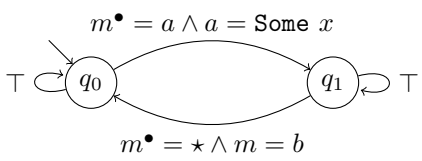

Figure 5: `Fifo(a,b)`

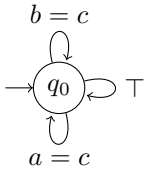

Figure 6: `Merger(a,b,c)`

We consider, for the next section, guard-covered and independent constraint automata. A naive way to run constraint automata in parallel is to find a solution to the composite guard $g_1 \wedge g_2$ as the union of a solution to g_1 and a solution to g_2. As variables occur free in g_1 and g_2, such approach may need to enumerate a large susbet of $\mathbb{D}$ before finding a satisfying assignment. Instead, we translate a declarative constraint (now expressed as a guard) into a set of operations on port structures. Thus, each constraint automaton may run those operations, in an order that preserves the dataflow, so that a resulting solution is found if it exists. We explain these operations in the next section.

2.2 Parallel product

We first review a few undesirable properties of the composition operation defined by Rule 1. First, observe that this static product is not *structure preserving*. Indeed, the conjunction of guards loses the original structure of the composition operands: the guard $g_1 \wedge g_2$ may decompose into several conjuncts, different from the original g_1 and g_2.

Second, the solution to the guard labeling a transition if obtained from an oracle that somehow solves the composite constraint. Such a solution may not be *realistic* in the sense that it does not emerge out of (and may not respect) the data flow through ports.

We present an algorithm that composes constraint automata while preserving the structure of the composition as a multiset of constraint automata, and that uses the dataflow at ports to find a global solution to the composite constraint.

We assume, for this section, that variables are partitionned into input and output variables. Let $A = (Q, \mathcal{V}, \rightarrow, Q_0)$ be a constraint automaton, and $\mathcal{V} = \mathcal{V}_I \cup \mathcal{V}_O$ be a partition of port variables in $\mathcal{V}$ such that $\mathcal{V}_I \cap \mathcal{V}_O = \emptyset$. A variable in $\mathcal{V}_I$ is an input variable, and a variable in $\mathcal{V}_O$ is an output variable. Every port is therefore either an input or an output (this distinction does not matter for memory variables).

2.2.1 Dataflow and constraint automata

We find a global solution to a composite constraint by locally performing some read/write operations on port structures. To give an idea of how solutions are constructed, we draw operations graphically as in Figure 7.

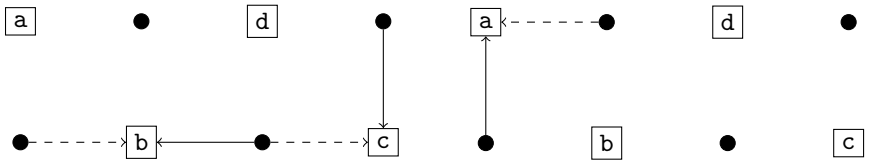

Figure 7: Two composite actions for the parallel product of the constraint automata `Prod(a)`, `Fifo(a, b)`, `Merger(b, c, d)`, and `Cons(c, d)`.

Ports are represented graphically with square boxes with letters. Constraint automata are black circles, and may do operations on ports with arrows: dashed arrows are *put* operations and full arrows are *take* operations.

On the left part of Figure 7, the bottom left black circle represents the `Fifo(a, b)` constraint automaton, as it puts (dashed arrow) some value in port b, while the merger takes (filled arrow) a data item from b, puts that data item in c, and the consumer gets the data item from c.

On the right part of Figure 7, the `Prod(a)` constraint automaton puts some data item on port a, and the `Fifo(a, b)` constraint automaton gets that data item from port a.

We identify two challenges in the dynamic construction of solutions as presented here. First, the order in which put and take operations are performed matters: a take operation on a port must always follow after a put places a value in that port; moreover, ports should be updated in the order of the dataflow (i.e., first b, then c for the example in Figure 7). As we show in the next subsection, we enforce such ordering at runtime.

Second, operations on ports may lead to some inconsistencies, resulting in failures. In case of a failure, the runtime needs to backtrack to explore alternatives.

With those two concerns in mind, we present an algorithm that performs the parallel product of constraint automata at runtime, i.e., it finds a solution for a composite transition if such solution exists.

2.2.2 Algorithm

Every port is a data structure with two buffers: a sync and a data buffer. For a port P, we use $P(sync) \in \{\bot, \top\}$ to refer to the sync buffer, and $P(data) \in D_\star$ to the data buffer of P. We require a total order among all ports, and we write $p < q$ when p is smaller than q. We assume that, in every constraint automaton, an input port is smaller than an output port to which it is synchronously related (i.e., the data from the input is directly put in the output). For instance, in the case of `Merger(a,b,c)`, a and b are input ports synchronously related to the output port c, so it must be that $a < c$ and $b < c$. Alternatively, a `Fifo(a,b)` imposes no such ordering constraints as a and b are not synchronously related. This property helps to solve the first challenge identified above, namely that the operations on ports are done in an order that respects the direction of data flow through ports.

We explain the main operations in Algorithm 1. For $i \in [1, n]$, let A_i be a guard covered and independent constraint automaton. We fix an initial state q_i for automaton A_i, with the initial assignments μ_i. The data and sync states of every port are also initialized to $\bot$ and $\star$, respectively. We require that every port as input for a constraint automaton is also used as an output for another constraint automaton, i.e., $\bigcup_{i\in[1,n]} \mathcal{V}_{i,I} = \bigcup_{i\in[1,n]} \mathcal{V}_{i,O}$.

The algorithm consists of an infinite loop, each iteration of which consists of an atomic update from a composite transition. The first step is to form a composite transition: given the state q_i of automaton A_i, one outgoing transition from s_i is selected for the iteration. Note that it is not required that each automaton performs a transition (as automata are assumed to be independent, see Theorem 2).

From each transition t_i with its guard g_i, the algorithm derives a series of operations O_i: if $p \in \mathcal{V}(g) \cap \mathcal{V}_I$, then $take(p) \in O_i$, if $p \in \mathcal{V}(g) \cap \mathcal{V}_O$, then $put(p) \in O_i$. These operations are later used to trigger changes in the buffers of their respective ports. Every guard that labels the transition is updated such that its memory variable m is replaced with the memory value $\mu_i(m)$. This list of operations is added to a global list opList.

The ordering of the optList ensures that operations on ports will correspond to data flow. As such, operations are sorted from operation of the smallest port to the highest port given the ordering relation on ports. This ensures that operations on input ports of channels are performed before operations on their respective output ports (by the assumption of ordering of ports of constraint automata). Next, the algorithm orders a $put(p)$ operation before a $take(p)$ operation for every p. This ensures that the automaton that takes the data

from a port will be able to read the value that has already been put in that port.

The *check* operation on opList returns *Fail* if there is not a $put(p)$ for every $take(p)$, and vice versa. The algorithm, thus, ensures the dataflow property that ports cannot retain data beyond a single atomic step.

Then, in sequence, for each operation o in the list, if the operation is a *take* operation, the *sync* buffer of the port is updated, the memory content of the port updates internal memory of the automaton, and port variables in g are updated with the new value. New memory assignments found in g then lead to updates of μ. If the operation is a *put* operation, the *data* buffer gets the value from the port variable.

Two possibilities exist for a round to be inconsistent: either the guard evaluates to false, or data fails to flow, i.e., a port produces without consuming, or consumes without producing, a value. In both cases, the algorithm resets the internal memory to the initial μ_i, and resets all port buffers. It then skips to the end of the loop and repeats the round with a new composite transition.

If no inconsitency is found in the guard or in the state of a port, each participating automaton moves to its next state, all memory variables for ports are reset, memories are updated, and all ports are reset.

Then the round ends and a new round starts, with updated memories and reset ports. We show in the next section how the runtime composition is implemented in Maude. We directly start with a set of operations for constraint automata, and therefore omit the translation from guards to list of operations.

Theorem 3 (soundness)**.** *The sequence of assignments of ports given by Algorithm 1 forms prefixes of an execution of the corresponding composite constraint automaton.*

3 Implementation in Maude

In this section, we present the details and evaluate an implementation in Maude of the runtime product of parallel constraint automata. Rewriting logic is a powerful framework for modeling concurrent systems [29, 28]. Moreover, implementations of rewriting logic, such as Maude [11], make system specifications both executable and analyzable. In our Maude implementation, the internal logic of a constraint automaton is expressed equationally, and the dynamics of its transitions is given as rewriting rules.

Our implementation in Maude enables two levels to make a preference-aware system more specific: by selecting a subset of best actions either at the constraint automaton level (i.e., which transition to offer first in the algorithm), or at the system level (i.e., which composite transition to select when several are possible).

Our rewriting specification of constraint automata has the following benefits. First, performing lazy composition keeps the representation of an interacting system small. Second, step-wise runtime composition renders our runtime

Algorithm 1 Parallel composition

Require: Constraint automata A_1, ..., A_n, in states q_1, ..., q_n, with initial memory $\mu_1, ..., \mu_n$; ports $p_1, ..., p_m$ with $p_i(sync) = \bot$ and $p_i(data) = \star$; and $\bigcup_{i\in[1,n]} \mathcal{V}_{i,I} = \bigcup_{i\in[1,n]} \mathcal{V}_{i,O}$.

repeat
- pick a new composite transition $T = [t_1, ..., t_k]$
- opList = []
- **for** $t_i \in [t_1, ..., t_k]$ **do**
  - convert the guard g_i on transition t_i into a list O_i of operations on ports
  - $g_i[m \mapsto \mu_i(m)]$
  - add O_i to opList
- **end for**
- order opList into a single list of operations
- **if** check(opList) = Fail **then**
  - continue
- **end if**
- **for** $o \in$ opList **do**
  - let A_i be the automaton that does operation o
  - **if** $o = take(p)$ **then**
    - $p(sync) \mapsto \top$
    - $\mu_i(p) \mapsto p(data)$
    - $g_i[p \mapsto \mu_i(p)]$
    - update μ with new values found from g
  - **else**
    - $p(data) \mapsto \mu_i(p)$
  - **end if**
- **end for**
- **if** one of the g_i is $\bot$ or one of the port is in inconsistent state **then**
  - μ_i is reinitialized to the initial μ_i
  - **for** p in all A_i **do**
    - $p(sync) \mapsto \bot$ and $p(data) \mapsto \star$
  - **end for**
  - continue
- **end if**
- **for** $t_i \in [t_1, ..., t_k]$ **do**
  - forall p in A_i, $\mu_i(p) \mapsto \star$
  - forall m in A_i, $\mu_i(m) \mapsto \mu_i(m^\bullet)$
  - $q_i \mapsto q_i'$ with q_i' the post state of t_i
- **end for**
- **for** p in all A_i **do**
  - $p(sync) \mapsto \bot$ and $p(data) \mapsto \star$
- **end for**

until True

Ensure: Consecutive data flow at ports $p_1, ..., p_m$ that satisfy a composite transition in $A_1, ..., A_n$.

framework modular, where run-time replacement of individual component automata becomes possible (as long as the update complies with some rules). Finally, our runtime framework more closely matches the architecture of a distributed framework, where entities are physically separate and no party may have access to the whole description of the entire system.

The running time of an execution may, in constrast, be slower than for the case of a static product. We discuss the challenges of our approach below.

Prerequisite The framework that we use has been developped for simulation and analysis of interacting *agents* with cyber-physical aspects [25]. In this current work, we use this framework to implement our parallel constraint automata. We implement both ports and constraint automata as instances of an *agent*, which is similar to an object:

```
[id : class | state ; flag ; actions]
```

where `id` is a unique identifier for the agent of type `Class`. Every agent has a `state`, that contains key/value pairs. A Boolean `flag` indicates whether or not an agent is available for composition (i.e., `flag` is `true`). The set of actions that an agent can perform in a round is given by `actions`, where an action in our case consists of *put* and *take* operations on ports.

A system of agents is simply a list of agents. The run of a system is given by the main rewrite rule:

```
crl[transition] : [sys]  => [sys']
    if agentsReady?(sys)  /\
       saAtom := getSysActions(sys) /\
       saComp := buildComposite(saAtom) /\
       p(actseq, sys') ; actseqs := kbestActions(saComp, k, sys)
```

where

- `sys` and `sys'` are multisets of agents before and after taking a composite transition, respectively;
- `agentsReady?` returns true if all agents have their `flag` set to `true`, i.e., ready to proceeed;
- `getSysActions` collects the list of actions for each agent in `sys`;
- `buildComposite` builds the list of composite actions (i.e., composite transitions); and
- `kbestActions` returns the k best acceptable composite actions, i.e., that do not bring the composition `sys` into an error state. The notion of *best* action is liberal in our framework (as it makes use of preferences), but not detailed in this paper. See [25] for more details.

We give, in Section 3.1, the definition of a port as an agent, and a channel as an agent.

3.1 Port and constraint automaton as agents

In order to achieve the goals set in the introduction, the runtime should accommodate for the concurrency inherent in the dataflow of a Reo circuit. Previous work [12] shows that constraint automata is not adequate to simulate the concurrent aspect of Reo models, and proposes multilabeled petri nets as an alternative. Instead, we implement the parallel extension of constraint automata described in the previous section in the rewriting framework we have developed in Maude.

Existing executable environments for Reo [8] focuse on compilation of input circuits into an executable code. A challenge not addressed in such work is the construction of a compositional runtime where each part of a Reo circuit can be compiled independently and run concurrently. This capability is important, for instance, to reconfigure a circuit during its execution. Also, in such a framework, one can keep the structure of a Reo circuit at runtime, mix different semantics for Reo channels (e.g., guarded commands, constraint automata, etc.), while allowing for simulation and verification through reachability queries.

We present, in the next paragraphs, the structure of our runtime environment to simulate, verify, and dynamically reconfigure Reo circuits. More precisely, we introduce two primitive sorts both encoded as agents in our framework: `PORT` and `CONNECTOR`. As shown in the previous subsection, a system specification is a multiset of agents, which is instantiated as a multiset of connectors and ports, through which data flow.

The code and examples discussed in this section are accessible at [24].

Port as resource A port is a point of synchronization in Reo, and its typical behavior is to atomically forward data from its input connector to its output connector. We fix a port identifier to be of the form `P(i)` where `i:Float` is a rational number. We later use the identifier of a port to define the operation of `linearization` and use the natural ordering of `Float` to order actions.

A port contains a structure with two buffers that implement the atomic passing of data from its input to its output connector. One buffer collects the data item that the input connector may *put*, and the other contains the request from the output connector to *take* a data item. Only when the two buffers are full should the port allow both `put(d)` and `take` actions.

Interface of a port:

```
[P(i) : Port | state ; flag ; action]
```

where:

- `i` is a float number that identifies a port. We use float in order to easily introduce new ports, and yet preserve a meaningful ordering over ports;
- `state` contains key/value pairs that collectively identify a state. For instance, the key `k("sync")` is mapped to a Boolean value that is `k("true")` when a `take` operation has been performed; the key `k("data")` contains

the data value given by some `put(d)` operation; and the key `k("blocked")` maps to `true` if the port has been blocked by another component;

- the `flag` will be set to `true` (as a port is always ready to proceed) and the `action` fields will be set to `empty`, as a port *reacts* to actions of other agents.

Connectors as agents We give three instances of primitive Reo connectors: a `SYNC`, a `FIFO`, and a `MERGER`. Other primitive connectors, such as a replicator, syncdrain, etc, are defined similarly. We show in the next section some more complex connectors, such as the alternator, defined as a composition of primitive connectors.

A `SYNC` agent has two ports on which it acts synchronously. The action of a `SYNC` agent consists of an atomic sequence of two actions: a `take` on its input port, and a `put` on its output port. The composite action succeeds if and only if the two parts involved in the action allow it to succeed, and the value *put* in the output port corresponds to the value *taken* from the input port.

A `FIFO` agent has two ports on which it acts in sequence. A `FIFO` agent has two actions, a `take` action that stores a data item from its input port into a memory, and a `put` action that outputs that data item through its output port. The `take` action succeeds only if the current memory cell is empty, and the `put` action succeeds only if the current memory cell is full. As a result, the `FIFO` agent alternates between taking a value from its input port, and putting that value into its output port.

A `MERGER` is a ternary connector that acts, for each of its input ports, as a synchronous channel with its output port. Moreover, the `MERGER` relates its two input ports with a relation of exclusion, i.e., the two input ports cannot fire at the same time. A `MERGER` with the list of ports `P(1), P(2), P(3)` has two actions: an action that forwards a data item from port `P(1)` to port `P(3)`, and an action that forwards a data item from port `P(2)` to port `P(3)`. These two actions are exclusive, as they cannot occur at the same time. Moreover, the merger always enables both actions, which raises some non-determinism at the system level (i.e., to chose which of the two actions to perform). Similarly to the `SYNC` channel, a `MERGER` agent instantiates the value for the `put` action at runtime, once the result of the `take` action is known. We use the `?` symbol to denote a value that will be put at a port but that is not already known (because it is not already taken from an input port).

A `PROD` and a `CONS` agent implement a producer and consumer, respectively. Each of these agents has a single port, on which it always performs, respectively, a `put` and a `take` action. The `PROD` agent puts natural numbers as values on its port, and increments the value when the `put` action succeeds. We use such canonical sequences of increasing natural numbers to verify certain firing properties of a Reo circuit. When a `put` action succeeds for a `PROD` agent, its state is updated to contain the message that has been sent in the `k("sent")` field. Similarly, when a `take` action succeeds for a `CONS` agent, its state is updated to contain the message that has been received in the `k("recv")` field.

Runtime composition As expected, runtime composition has several drawbacks: if there are n automata to compose, each with one transition, there are 2^n possible combination of composite transitions. This is easily understood as a tuples of size n, where the i^{th} location tells whether the transition has been selected or not (i.e., 1 means that the transition of the i^{th} automata is in the product). The enumeration of all possibilities, therefore, leads to 2^n cases.

Note that we can prune some compositions that are not *well-formed*. By construction, we know that every `put(d)` action on a port `P(i)` must be in the atomic set of a `take` action on that same port. Thus, any combination that has one but not the other can already be removed.

The next operation that speeds up runtime composition is called *linearization*. This operation takes a list of transitions (`aSeq`), and reorders it so that the runtime checks and applies each transition in sequence. To do so, we use the name of the port, as a float number, to order the transitions: action on ports with a smaller port numbers will be run first at runtime. As we require that a channel has increasing port numbers, we ensure that, if possible, data will flow from its input to its output.

In Maude, the implementation of linearization is done as follows:

```
ceq linearization(aSeq) = orderActionPort(seq) linearization(aSeq')
   if  seq := getActionsFromPort(aSeq, getSmallestPort(aSeq))  /\
       aSeq' := filterOut(aSeq, makeSet(seq)) /\ aSeq =/= nil .
eq orderActionPort(aSeq) = putValue(aSeq) takeValue(aSeq) .it
```

where

- `orderActionPort` orders, from right to left, actions on the same port. The ordering for a given port places first *put* actions, then *take* actions so that the *take* action will eventually have the data from a preceding *put* action;
- `linearization` is recursively applied on the remaining actions.

3.2 Execution and analysis

We show the utility of our framework by example applications within three different categories: verification with reachability queries, normalisation of connectors, and dynamic reconfiguration. The fact that all three use cases can be investigated within the same framework is also a benefit compared with existing works.

The first category is the most intuitive one, as it provides a way to simulate Reo circuits. Our example in this category is the alternator connector.

The second category uses the algebraic nature of Maude: we equationally define the equivalence of circuits, and therefore simplify a system to a smaller one. For instance, the composition of two sync channels can be rewritten as a single sync channel as the two circuits are behaviorally equivalent.

The third category of examples explores the use of rewrite rules to do dynamic reconfiguration: a new component is added to the system at runtime each

time a reconfigurable rule is triggered. As an example in this category, we show how to model an unbounded fifo.

Dataflow properties We first use our framework to verify dataflow properties of a binary alternator connector. A binary alternator has two input ports and an output port. In our example, we connect each input port to a producer and the output port to a consumer process. The behavior of a binary alternator is as follows: it synchronizes its two input ports to ensure that both producers connected to these ports have data items to offer. It then places the data items from its input ports in its output port in a fixed, predefined order; thus, what the alternator produces through its output port is an alternating sequence of the data items that it obtains from its two input ports. A property of the alternator is therefore that no two data items of one producer is observed in its output next to each other.

We give the specification of the alternatorScenario, and then check some of its properties. The alternatorScenario consists of two `Prod` agents that communicate with a `ConsList` agent through an alternator protocol:

```
alternator =
[PortList(1.0, 1.5, 1.8, 2.0, 2.5, 2.8, 3.0, 3.5, 4.0)
 Prod(P(1.0)) Prod(P(2.0)) ConsList(P(4.0))
 Replicator(P(1.0), P(1.5), P(1.8))
 SyncDrain(P(1.5), P(2.5))
 Sync(P(1.8), P(3.0))
 Merger(P(3.0), P(3.5), P(4.0))
 Replicator(P(2.0), P(2.5), P(2.8))
 Fifo(P(2.8), P(3.5), empty)]
```

All the channels, with the exception of `Prod` and `ConsList` in the above are primitives that constitute the internals of an alternator. The `ConsList` agent records the sequence of data received. We assume that `Prod`$(P(i))$ sends data $nd(i)$ to its port (where `nd` stands for *natural data* to denote natural numbers).

We show two search commands on an alternator: one that looks for consecutive data in the consumer that are different (i.e., alternating), and one that looks for consecutive data that come from the same agent.

The execution of the following search result returns no solution in a reasonable amount of time, leading to the conclusion that no consecutive data of the same producer can be output from the alternator:

```
search [1] in SCENARIO : alternator =>* [sys::Sys
  [ConsList(P(4.0)) : Consumer |
    M2::MapKD, k("data") |-> nd(i::Nat) nd(j::Nat) ; false ; null]]
    such that i::Nat == j::Nat = true .
```

One solution is found, however, when searching two consecutive data items coming from different producers. The following search query asks to find a rewriting sequence that leads the consumer observe two consecutive data with different values.

```
search [1] in SCENARIO : alternator =>* [sys::Sys
  [ConsList(P(4.0)) : Consumer |
```

```
    M2::MapKD, k("data") |-> nd(i::Nat) nd(j::Nat) ; false ; null]]
    such that i::Nat =/= j::Nat = true .
```

Simplification of circuits If one proves that the behavior of two circuits are the same, one would like to simplify the system term to the smaller and simpler of the two specifications. For instance consider the composition of two consecutive sync channels:

```
  eq init = [
    [Prod(P(1.0)) : Producer | k("data") |-> 1.0 ; false ; null]
    [Cons(P(3.0)) : Consumer | k("data") |-> nodata; false ; null]
    [P(1.0) : Port | state ; false ; null]
    [P(2.0) : Port | state ; false ; null]
    [P(3.0) : Port | state ; false ; null]
    [Sync(P(1.0), P(2.0)) : Channel | state ; false ; null]
    [Sync(P(2.0), P(3.0)) : Channel | state ; false ; null]
  ] .
```

The above expression can provably be simplified by replacing the two `Sync` channels with a single `Sync` channel, and removing the intermediate port. We can give the following equation in Maude:

```
  var i j k : Float .

  ceq [sys] = [
      [P(i) : Port | state ; false ; null]
      [P(k) : Port | state ; false ; null]
      [Sync(P(i), P(k)) : Channel | state ; false ; null] sys']
    if
 [P(i) : Port | state ; false ; null]
 [P(j) : Port | state ; false ; null]
 [P(k) : Port | state ; false ; null]
 [Sync(P(i), P(j)) : Channel | state ; false ; null]
 [Sync(P(j), P(k)) : Channel | state ; false ; null] sys' :=  sys .
```

Then, the reduction of the term `init` is the new term:

```
init = [
 [Prod(P(1.0)) : Producer | k("data") |-> 1 ; false ; null]
 [P(1.0) : Port | state ; true ; null]
 [P(3.0) : Port | state ; true ; null]
 [Cons(P(3.0)) : Consumer | k("data") |-> nodata ; false ; null]
 [Sync(P(1.0), P(3.0)) : Channel | state ; false ; null]]
```

Dynamic updates As we showed earlier, a Reo circuit can be simulated, and the state space can be explored within our framework implemented in Maude. We now present an additional feature, namely the possibility to dynamically reconfigure a circuit. Dynamic reconfiguration has already been explored in some previous tools supporting simulation of Reo circuit [21]. However, that implementation required an ad hoc plugin to interface the main environment. In our case, the simulation, verification, and reconfiguration are all done in the same environment, expressed with rewriting logic.

We show, as a use case, how we can use our bounded `Fifo` channel (with capacity of 1) to implement an unbounded fifo that triggers a reconfiguration at the system level each time it receives a data. We add two rewrite rules together with the specification of the unbounded fifo: `add-buffer` and `rem-buffer`.

The first rewrite rule adds a buffer whenever the `UnboundedFifo` is full:

```
crl [add-buffer] :
 [UnboundedFifo(P(n1), P(n2)) : Channel | M, k("state") |-> nd(1) ;
      false ; null] sys] =>
  [[P((n1 + n2) / 2.0) : Port | defaultState ; false ; null]
  [UnboundedFifo(P(n1), P((n1 + n2) / 2.0)) : Channel | k("data")
      |-> nodata, k("state") |-> nd(0) ; false ; null]
  [Fifo(P((n1 + n2) / 2.0), P(n2)) : Channel | M, k("state") |-> nd
      (1) ; false ; null] sys] .
   if [Fifo(P(n2), P(n3)) : Channel | M, k("state") |-> nd(1) ;
       false ; null] sys' := sys .
```

Then, the initial term `init` given by a `Prod`, a `Cons`, and an `UnboundedFifo` leads to the following system after 9 rewrites:

```
Maude> cont 9 .
    rewrites: 8105 in 3ms cpu (3ms real) (2041047 rewrites/second)
    result Global: [
    [Prod(P(1.0)) : Producer | k("data") |-> nd(1), k("msg") |-> nd
        (1) ; false ; null]
    [P(1.0) : Port | k("data") |-> nodata, k("sync") |-> bd(false)
        ; true ; null]
    [P(1.5) : Port | k("data") |-> nodata, k("sync") |-> bd(false)
        ; true ; null]
    [P(2.0) : Port | k("data") |-> nodata, k("sync") |-> bd(false)
        ; true ; null]
    [Cons(P(2.0)) : Consumer | k("data") |-> nd(1) ; false ; null]
    [Fifo(P(1.5), P(2.0)) : Channel | k("data") |-> nodata, k("
        state") |-> nd(0) ; false ; null]
    [UnboundedFifo(P(1.0), P(1.5)) : Channel | k("data") |-> nd(1),
         k("state") |-> nd(1) ; false ; null]]
```

As we can see, an intermediate empty fifo has been added to the system, which will then take the value from the `UnboundedFifo`, and forward it to the `Cons`. If `Prod` produces values faster than `Cons` consumes them, the `UnboundedFifo` will interleave new `Fifo`, in order, to implement the unbounded behavior.

The second rule to *remove* empty buffers is:

```
  rl [rem-buffer] :
    [[UnboundedFifo(P(n1), P((n1 + n2) / 2.0)) : Channel | M ;
        false ; null]
     [P((n1 + n2) / 2.0) : Port | k("data")|-> nodata, k("sync")
         |-> bd(false) ; true ; null]
     [Fifo(P((n1 + n2) / 2.0), P(n2))          : Channel | k("data
         ") |-> nodata, k("state") |-> nd(0) ; false ; null] sys]
         =>
    [[UnboundedFifo(P(n1), P(n2))            : Channel | M ; false
        ; null] sys] .
```

As shown after running the search command

```
search [1] init =>*
```

```
[[Prod(P(1.0)) : Producer | k("data") |-> nd(1), k("msg") |->
    nd(1) ; false ; null]
 [P(1.0) : Port | k("data") |-> nodata, k("sync") |-> bd(false)
      ; true ; null]
 [P(2.0) : Port | k("data") |-> nodata, k("sync") |-> bd(false)
      ; true ; null]
 [Cons(P(2.0)) : Consumer | k("data") |-> nd(1) ; false ; null]
 [UnboundedFifo(P(1.0), P(2.0)) : Channel | k("data") |-> nd(1)
     , k("state") |-> nd(1) ; false ; null]] .
```

the added buffer has been removed by the *rem-buffer* rewrite rule when it was emptied by the consumer. Thus, we successfully implemented an unbounded fifo by using two additional rewrite rules.

4 Conclusion

Reo is a powerful channel-based language that models coordination among processes. Constraint automata is an operational semantics for Reo channels, that supports model checking techniques. Unfortunately, constraint automata are not suitable for simulation and dynamic reconfiguration, and their (static) product violates architectural fidelity: the requirement to perform the product statically not only creates a state space explosion, it also loses the internal structure of composition of the components in a system. We presented in this paper the PCA model: a modified version of constraint automata that preserves architectural fidelity and enables simulation, verification, and dynamic reconfiguration of Reo circuit. We demonstrated the utility of the PCA with an implementation in Maude and a series of examples.

Given the capabilities of the execution environment presented in the paper, the algebraic nature of the specification of Reo channels in Maude opens new avenues for new connector. For instance, *physical* aspects of components can be algebraically encoded, and would supplement Reo's existing exogeneous coordination primitives. Thus, time sensitive circuits could be simulated, verified, and could profit from the possibility of dynamic reconfiguration. A first step has been taken in this direction in the PhD thesis [23]. We leave as future work the improvements to our current Maude implementation to support more examples (e.g., cyber-physical Reo circuits), to optimize further the search queries for verification, and to take full advantage of the rewriting features provided by Maude for, e.g., dynamically reconfigurable circuits.

5 Acknowledgement

The authors are extremely grateful to Carolyn Talcott for her immensely valuable guidance and input during the development of the Maude framework that underlies the PCA implementation presented in this paper. Any remaining errors of fact, in implementation, or presentation are, of course, the authors'.

References

[1] Luca Aceto, Augusto Burgueño, and Kim Guldstrand Larsen. "Model Checking via Reachability Testing for Timed Automata". In: *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Edited by Bernhard Steffen. Volume 1384. Lecture Notes in Computer Science. Springer, 1998, pages 263–280. DOI: 10.1007/BFB0054177.

[2] Farhad Arbab. "Reo: a channel-based coordination model for component composition". In: *Math. Struct. Comput. Sci.* 14.3 (2004), pages 329–366. DOI: 10.1017/S0960129504004153.

[3] Farhad Arbab, Tom Chothia, Rob van der Mei, Sun Meng, Young-Joo Moon, and Chrétien Verhoef. "From Coordination to Stochastic Models of QoS". In: *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*. Edited by John Field and Vasco Thudichum Vasconcelos. Volume 5521. Lecture Notes in Computer Science. Springer, 2009, pages 268–287. DOI: 10.1007/978-3-642-02053-7_14.

[4] Farhad Arbab, Natallia Kokash, and Sun Meng. "Towards Using Reo for Compliance-Aware Business Process Modeling". In: *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*. Edited by Tiziana Margaria and Bernhard Steffen. Volume 17. Communications in Computer and Information Science. Springer, 2008, pages 108–123. DOI: 10.1007/978-3-540-88479-8_9.

[5] Farhad Arbab and Sun Meng. "Synthesis of Connectors from Scenario-Based Interaction Specifications". In: *Component-Based Software Engineering*. Edited by Michel R. V. Chaudron, Clemens Szyperski, and Ralf Reussner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 114–129. ISBN: 978-3-540-87891-9.

[6] Christel Baier, Tobias Blechmann, Joachim Klein, Sascha Klüppelholz, and Wolfgang Leister. "Design and Verification of Systems with Exogenous Coordination Using Vereofy". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Edited by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 97–111.

[7] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. "Modeling component connectors in Reo by constraint automata". In: *Sci. Comput. Program.* 61.2 (2006), pages 75–113. DOI: 10.1016/J.SCICO.2005.10.008.

[8] Kasper Dokter Benjamin Lion. *The Reo Language repository*. https://github.com/ReoLanguage/Reo. 2017.

[9] Dave Clarke and José Proença. "Coordination via Interaction Constraints I: Local Logic". In: *Proceedings 2nd Interaction and Concurrency Experience: Structured Interactions, ICE 2009, Bologna, Italy, 31st August 2009*. Edited by Filippo Bonchi, Davide Grohmann, Paola Spoletini, and Emilio Tuosto. Volume 12. EPTCS. 2009, pages 17–39. DOI: 10.4204/EPTCS.12.2.

[10] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. "Channel-based coordination via constraint satisfaction". In: *Sci. Comput. Program.* 76.8 (2011), pages 681–710. DOI: 10.1016/J.SCICO.2010.05.004.

[11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Volume 4350. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-71940-3. DOI: 10.1007/978-3-540-71999-1.

[12] Kasper Dokter. "Multilabeled Petri Nets". In: *Formal Aspects of Component Software - 16th International Conference, FACS 2019, Amsterdam, The Netherlands, October 23-25, 2019, Proceedings*. Edited by Farhad Arbab and Sung-Shik Jongmans. Volume 12018. Lecture Notes in Computer Science. Springer, 2019, pages 106–126. DOI: 10.1007/978-3-030-40914-2_6.

[13] Kasper Dokter and Farhad Arbab. "Rule-Based Form for Stream Constraints". In: *Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings*. Edited by Giovanna Di Marzo Serugendo and Michele Loreti. Volume 10852. Lecture Notes in Computer Science. Springer, 2018, pages 142–161. DOI: 10.1007/978-3-319-92408-3_6.

[14] S.-S. T.Q. Jongmans and F. Arbab. "Overview of Thirty Semantic Formalisms for Reo". In: *Scientific Annals of Computer Science* 22.1 (2013), pages 201–251. DOI: 10.7561/SACS.2012.1.201.

[15] S.-S.T.Q. Jongmans and F. Arbab. "Centralized coordination vs. partially-distributed coordination with Reo and constraint automata". In: *Science of Computer Programming* 160 (2018). Fundamentals of Software Engineering (selected papers of FSEN 2015), pages 48–77. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2017.06.004.

[16] Sung-Shik Jongmans. "Automata-Theoretic Protocol Programming: Parallel computation, threads and their interaction, optimized compilation [at a] high level of abstraction". English. PhD thesis. Netherlands: Leiden University, Mar. 2016.

[17] Sung-Shik T. Q. Jongmans and Farhad Arbab. "PrDK: Protocol Programming with Automata". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Edited by Marsha Chechik and Jean-François Raskin. Volume 9636. Lecture Notes in Computer Science. Springer, 2016, pages 547–552. DOI: 10.1007/978-3-662-49674-9_33.

[18] Sung-Shik T. Q. Jongmans, Tobias Kappé, and Farhad Arbab. "Constraint automata with memory cells and their composition". In: *Sci. Comput. Program.* 146 (2017), pages 50–86. DOI: 10.1016/J.SCICO.2017.03.006.

[19] Sascha Klüppelholz and Christel Baier. "Symbolic model checking for channel-based component connectors". In: *Sci. Comput. Program.* 74.9 (2009). DOI: 10.1016/J.SCICO.2008.09.020.

[20] Natallia Kokash, Christian Krause, and Erik P. de Vink. "Reo + mCRL2: A framework for model-checking dataflow in service compositions". In: *Formal Aspects Comput.* 24.2 (2012), pages 187–216. DOI: 10.1007/S00165-011-0191-6.

[21] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. "Modeling dynamic reconfigurations in Reo using high-level replacement systems". In: *Sci. Comput. Program.* 76.1 (2011), pages 23–36. DOI: 10.1016/J.SCICO.2009.10.006.

[22] François Laroussinie and Kim Guldstrand Larsen. "Compositional Model Checking of Real Time Systems". In: *CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*. Edited by Insup Lee and Scott A. Smolka. Volume 962. Lecture Notes in Computer Science. Springer, 1995, pages 27–41. DOI: 10.1007/3-540-60218-6_3.

[23] Benjamin Lion. "An algebra for interaction of cyber-physical components". PhD thesis. June 2023.

[24] Benjamin Lion. *The Maude Implementation of Reo*. https://benjaminlion.fr/reo.zip. 2024.

[25] Benjamin Lion, Farhad Arbab, and Carolyn L. Talcott. "A Rewriting Framework for Interacting Cyber-Physical Agents". In: *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part III*. Edited by Tiziana Margaria and Bernhard Steffen. Volume 13703. Lecture Notes in Computer Science. Springer, 2022, pages 356–372. DOI: 10.1007/978-3-031-19759-8_22.

[26] Sun Meng and Farhad Arbab. "QoS-Driven Service Selection and Composition Using Quantitative Constraint Automata". In: *Fundam. Informaticae* 95.1 (2009), pages 103–128. DOI: 10.3233/FI-2009-144.

[27] Sun Meng, Farhad Arbab, and Christel Baier. "Synthesis of Reo circuits from scenario-based interaction specifications". In: *Sci. Comput. Program.* 76.8 (2011), pages 651–680. DOI: 10.1016/J.SCICO.2010.03.002.

[28] José Meseguer. "Conditioned Rewriting Logic as a United Model of Concurrency". In: *Theor. Comput. Sci.* 96.1 (1992), pages 73–155. DOI: 10.1016/0304-3975(92)90182-F.

[29] José Meseguer. "Twenty years of rewriting logic". In: *J. Log. Algebraic Methods Program.* 81.7-8 (2012), pages 721–781. DOI: 10.1016/j.jlap.2012.06.003.

[30] Young-Joo Moon, Alexandra Silva, Christian Krause, and Farhad Arbab. "A compositional model to reason about end-to-end QoS in Stochastic Reo connectors". In: *Sci. Comput. Program.* 80 (2014), pages 3–24. DOI: 10.1016/J.SCICO.2011.11.007.

[31] José Proença, Dave Clarke, Erik P. de Vink, and Farhad Arbab. "Dreams: a framework for distributed synchronous coordination". In: *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*. Edited by Sascha Ossowski and Paola Lecca. ACM, 2012, pages 1510–1515. DOI: 10.1145/2245276.2232017.