Compositional Semantics of an Actor-Based Language Using Constraint Automata

Marjan Sirjani[1,2], Mohammad Mahdi Jaghoori[2], Christel Baier[3], and Farhad Arbab[4,5]

[1] University of Tehran, Tehran, Iran
[2] IPM School of Computer Science, Tehran, Iran
[3] University of Bonn, Bonn, Germany
[4] CWI, Amsterdam, Netherlands
[5] Leiden University, Leiden, Netherlands
msirjani@ut.ac.ir, jaghoori@mehr.sharif.edu, baier@cs.uni-bonn.de
farhad@cwi.nl

Abstract. Rebeca is an actor-based language which has been successfully applied to model concurrent and distributed systems. The semantics of Rebeca in labeled transition system is not compositional. In this paper, we investigate the possibility of mapping Rebeca models into a coordination language, Reo, and present a natural mapping that provides a compositional semantics of Rebeca. To this end, we consider reactive objects in Rebeca as components in Reo, and specify their behavior using constraint automata as black-box components within Reo circuits. Modeling coordination and communication among reactive objects as Reo circuits, and the behavior of reactive objects as constraint automata, provides a compositional semantics for Rebeca. Although the result is a compositional model, its visual representation in Reo shows very well that it still reflects the tight coupling inherent in the communication mechanism of object-based paradigms, whereby the real control and coordination is built into the code of the reactive objects themselves. We describe an alternative design that overcomes this deficiency. This illustrates the differences between objects and components, and the challenges in moving from object-based to component-based designs.

Keywords: actor model, Compositional semantics, Rebeca, Reo, Constraint Automata.

1 Introduction

Managing large and complex systems requires techniques that support reusability and modifiability [1]. In general, compositionality allows one to master both the complexity of the design and verification of software models. Having a compositional semantics for a modeling language allows us to construct a model from its sub-models and reuse the already derived semantics of the sub-models. Compositional construction and verification can be exploited effectively only when the model is naturally decomposable [2], and there is no general approach for

P. Ciancarini and H. Wiklicky (Eds.): COORDINATION 2006, LNCS 4038, pp. 281–297, 2006.

decomposing a system into components [3]. Different researchers have worked on composing specifications and verifying their properties [4, 5, 6]. In this paper, we build up a compositional semantics for an actor-based language, using a component-based language and taking advantage of its compositional semantics. In this way we can use our object-based Java-like modeling language which is familiar for software engineers, while benefitting from the component-based paradigm to build models from their sub-models.

Rebeca (*Reactive Objects Language*) is an actor-based language with a formal foundation, presented in [7, 8, 9]. A model in Rebeca consists of a set of reactive objects (called *rebecs*) which are concurrently executing and asynchronously communicating. Rebeca can be considered as a reference model for concurrent computation, based on an operational interpretation of the actor model [10, 11]. It is also a platform for developing object-based concurrent systems in practice. Formal verification approaches are used to ensure correctness of concurrent and distributed systems. The Rebeca Verifier tool, as a front-end tool, translates Rebeca code into languages of existing model-checkers, allowing verification of their properties [12, 13]. There is also an ongoing project on developing a direct model checker for Rebeca using state space reduction techniques [14, 15].

The Rebeca semantics, expressed in LTS (Labeled Transition System) [7, 8] is not compositional. We cannot construct the semantics of the total model by composing the semantics of each rebec used to construct the model. The compositional verification approach proposed in [7, 9] is based on decomposing a closed Rebeca model and not composing the rebecs as the components of a model.

Reo [16, 17] is an exogenous coordination model wherein complex coordinators, called connectors are compositionally built out of simpler ones. The atomic connectors are a set of user-defined point-to-point *channels*. Reo is based on the foundation model of Abstract Behavior Types (ABT), as a higher level alternative to Abstract Data Types (ADT), which serve as the foundation of object oriented languages [18]. Reo can be used as a *glue language* for compositional construction of connectors that orchestrate component instances in a component based system.

In this paper, we investigate the possibility of mapping Rebeca models into Reo and propose a natural mapping that provides a compositional semantics of Rebeca. As reactive objects (rebecs) are encapsulated and loosely coupled modules in Rebeca, we consider them as components in a coordination language. Modeling the coordination and communication mechanisms between rebecs can be done by Reo circuits, and the behavior of each rebec is specified by constraint automata [19] as a black-box component within the Reo circuit.

In [20] and [21] a component-based version of Rebeca is proposed where the components act as wrappers that provide higher level of abstraction and encapsulation. The main problem in constructing a component in this object-based configuration is the rebec-to-rebec communication and the need to know the receiver names. In [20] and [21] components are sets of rebecs and the communication between components is via broadcasting anonymous and asynchronous

messages. In this paper, we use the coordination language Reo and model each rebec as a component. The rebec-to-rebec communication remains the main problem in exploiting the reusability provided by our compositional semantics. The semantics of Rebeca in Reo demonstrates this problem very well. We propose a solution based on the behavior of synchronous channels in Reo, the interleaving nature of concurrency in Rebeca models, and the fact that, in this case, there is only one message sent in an atomic step. Hence, the work in this paper is our first successful attempt to build-up components out of reactive objects without changing the semantics of Rebeca.

Another interesting outcome of our mapping is to clearly show the problems in moving from an object-based model to a component-based model of the kind proposed by Reo. The components that we construct out of reactive objects are not really amenable to external coordination control provided by the glue code. We cannot simply change the coordination glue code and expect another execution pattern independent from the behavior of the rebecs. The coupling inherent in the message passing mechanism will also be reflected in the Reo circuitry representing their communication. We show that in this case the glue code will grow in size and complexity as the system evolves. We then propose a solution for the special case of Rebeca.

Organization of the Paper. In Section 2, we provide a brief overview of Rebeca and a Rebeca model as an example which we also use in Section 7. Reo is described in Section 3, and our mapping of Rebeca to Reo is explained in Section 4. Constraint automata are used to build the compositional semantics of Reo. In Section 5 we describe their extended form of parameterized constraint automata which we use in this paper. In Section 6 we describe our algorithm for generating parameterized constraint automata out of Rebeca code. Section 7 shows a case study. Section 8 is a short conclusion and a view of our future work.

2 Rebeca: An Actor-Based Language

Rebeca models consist of concurrently executing *reactive objects*, called rebecs. Rebecs are encapsulated objects, with no shared variables, which can communicate only by asynchronous message passing. Each rebec is instantiated from a *reactive class* and has a single thread of execution; it also has an unbounded buffer, called a queue, for arriving messages. Computation takes place by message passing and execution of the corresponding methods of messages. Each message specifies a unique method to be invoked when the message is serviced. In this paper we abstract from dynamic object creation and dynamic topology, both of which are present in Rebeca models.

The operational semantics of Rebeca is defined using as a labeled transition system, a quadruple of a set of states (S), a set of labels (L), a transition relation on states (T), and a set of initial states of the system (s_0), $M = (S, L, T, s_0)$, where we have the followings (for a more detailed formal definition refer to [7]): The state space of the model is $\prod_{i=1}^{n}(S_i \times q_i)$, where each S_i denotes the local state of rebec r_i consisting of a valuation that maps each local field variable to a

value of the appropriate type; and the inbox q_i, an *unbounded* buffer that stores all incoming messages for rebec r_i in a FIFO manner.

The set of action labels L is the set of all possible message calls in the given model; such calls cause the processing of those messages that are part of the target rebec (if it provides the corresponding message server).

A triple $(s, l, s') \in S \times L \times S$ is an element of the transition relation T iff

- in state s there is some r_i such that l is the first message in the inbox q_i, l is of the form $\langle sender, receiver, msg \rangle$, where *sender* is the rebec identifier of the requester (implicitly known by the receiver), *receiver* is the rebec identifier of r_i (receiver rebec), and *msg* is the name of the method m of r_i which is invoked;
- state s' results from state s through the atomic execution of two activities: first, rebec r_i deletes the first message l from its inbox q_i, second, method m is executed in state s. The latter may add requests to rebecs' inboxes (by sending messages), change the local state (by assignments), and/or create new rebecs;
- if new rebecs are created in the invocation of m, then the state space S *expands dynamically*, which is out of the scope of this paper.

Clearly, the execution of the above methods relies implicitly on a standard semantic for the imperative code in the body of method m. Regarding the *infinite* behavior of our semantics, communication is assumed to be fair [11]: all the sent messages eventually reach their respective inboxes and will eventually be serviced by the corresponding rebec. The initial state s_0 is the one where each rebec has its `initial` message as the sole element in its inbox.

We use a simple example of trains and a controller to show a Rebeca model and also the mapping algorithm further in Section 7. Consider a bridge with a track where only one train can pass at a time. There are two trains, entering the bridge in opposite directions. A bridge controller uses red lights to prevent any possible collision of trains, and guarantees that each train will finally enter the bridge assuming that the trains pass the bridge after entering it.

Figure 1 shows the Rebeca code for the bridge controller example. There are two reactive classes, one for the bridge controller and one for the trains. The numbers in front of each reactive class name show the length of the queue of the rebecs instantiated from that class. For model checking purposes we need a bound on the queue lengths. The bridge controller uses its state variables to keep the value of the red lights on each side, and has flags to know whether or not a train is waiting on each side of the bridge. In the initial state, rebecs have their initial messages as the only message in their queues.

3 Reo: A Coordination Language

Reo is a model for building component connectors in a compositional manner [16, 17]. Reo offers a compositional approach to defining component connectors. Reo *connectors* (also called *circuits*) are constructed in the same spirit

```
reactiveclass BridgeController(5) {
  knownobjects{Train t1; Train t2;}
  statevars {
      boolean isWaiting1; boolean isWaiting2;
      boolean signal1;    boolean signal2;
  }
  msgsrv initial() {
      signal1 = false; isWaiting1 = false;
      signal2 = false; isWaiting2 = false;
  }
  msgsrv Arrive() {
      if (sender == t1) {
          if (signal2 == false) {
              signal1 = true;
              t1.YouMayPass();
          } else { isWaiting1 = true; }
      } else {
          if (signal1 == false) {
              signal2 = true;
              t2.YouMayPass();
          } else { isWaiting2 = true; } }
  }
  msgsrv Leave() {
      if (sender == t1) {
          signal1 = false;
          if (isWaiting2) {
              signal2 = true;
              t2.YouMayPass();
              isWaiting2 = false; }
      } else {
          signal2 = false;
          if (isWaiting1) {
              signal1 = true;
              t1.YouMayPass();
              isWaiting1 = false; } }
  }
}

reactiveclass Train(3) {
  knownobjects{BridgeController controller;}
  statevars { boolean onTheBridge; }
  msgsrv initial() {
      onTheBridge = false;
      self.Passed();
  }
  msgsrv YouMayPass() {
      onTheBridge = true;
      self.Passed();
  }
  msgsrv Passed() {
      onTheBridge = false;
      controller.Leave();
      self.ReachBridge();
  }
  msgsrv ReachBridge() {
      controller.Arrive();
  }
}
main {
    Train train1(theController);
    Train train2(theController);
    BridgeController theController
                        (train1, train2);
      }
```

Fig. 1. Rebeca Model for a Bridge Controller

as logic and electronics circuits: take basic elements and connect them. Basic connectors in Reo are *channels*. Each channel has exactly two ends, which can be a *sink* end or a *source* end. A *sink* end is where data flows out of a channel, and a *source* end is where data flows into a channel. It is possible for the ends of a channel to be both sinks or both sources. Reo places no restriction on the behavior of a channel. This allows an open-ended set of different channel types to be used simultaneously together in Reo, each with its own policy for synchronization, buffering, ordering, computation, data retention/loss, etc. For our purpose to model Rebeca models, we need a small set of basic channels, which we define later (in Figure 5).

Channels are connected to make a circuit. Connecting (or *joining*) channels is putting channel ends together in a *node*. So, a *node* is a set of coincident channel ends. The semantics of a node is as follows.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a *replicator*. A component can obtain data items, by an input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic *merger*. A mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends.

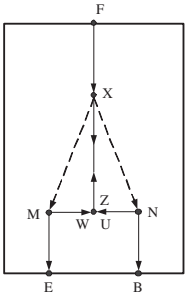

Fig. 2. Exclusive Router in Reo

Figure 2 shows a Reo connector, an exclusive router, which we call *Xrouter*. Here, we use it to show the visual syntax for presenting Reo connector graphs and some frequently useful channel types. This circuit is also used to model Rebeca in Reo. The enclosing thick box in this figure represents *hiding*: the topologies of the nodes (and their edges) inside the box are hidden and cannot be modified. It yields a connector with a number of input/output *ports*, represented as nodes on the border of the bounding box, which can be used by other entities outside the box to interact with and through the connector.

The simplest channels used in these connectors are synchronous (*Sync*) channels, represented as simple solid arrows (like edges FX and MW in Figure 2). A Sync channel has a source and a sink end, and no buffer. It accepts a data item through its source end iff it can simultaneously dispense it through its sink. A lossy synchronous (*LossySync*) channel is similar to a Sync channel, except that it always accepts all data items through its source end. If it is possible for it to simultaneously dispense the data item through its sink (e.g., there is a take operation pending on its sink) the channel transfers the data item; otherwise the data item is lost. LossySync channels are depicted as dashed arrows, e.g., XM and XN in Figure 2. Another channel is the synchronous drain channel (*SyncDrain*), whose visual symbol appears as the edge XZ in Figure 2. A *SyncDrain* channel has two source ends. Because it has no sink end, no data value can ever be

obtained from this channel. It accepts a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well. All data accepted by this channel are lost.

Two channels that are used in modeling Rebeca but are not included in the *Xrouter* circuit, are *FIFO* and *Filter* channels. We define *FIFO* as an unbounded asynchronous channel where data can flow in unboundedly from its source and flow out of its sink, if its buffer is not empty; input and output cannot take place simultaneously when the buffer is empty. Figure 5.a in Section 5 shows the Reo notation (and the constraint automaton) for a 1-bounded *FIFO* channel. *Filter* is a channel with a corresponding data pattern. It lets the data that match with the pattern pass and loses all other data. A *Filter* channel and its constraint automaton are shown in Figure 5.b.

4 Rebecs as Components in Reo

To model Rebeca using Reo, we can consider each rebec as a black-box component, and model the coordination and communication among the rebecs as Reo circuits. To model this coordination, we use an *Xrouter* which passes the control to each rebec nondeterministically. Communication takes place by asynchronous message passing which is modeled by FIFO and filter channels in Reo.

Each rebec starts its execution by receiving a *start* signal, and sends an *end* signal at its end. The behavior of a rebec as a component is to take a message from its message queue upon receiving the *start* signal through its start port, execute the corresponding message server, and send an *end* signal through its end port. The coordination, which is modeled by interleaved execution of rebecs, is handled by an *Xrouter* which passes the *start* signal to one and only one rebec, waits until it receives an *end* signal, and passes the *start* signal again, guaranteeing the atomic execution of each method according to the semantics of Rebeca in [7]. This loop is repeated by *Xrouter*, and sending the signals is done by a nondeterministic choice. The Reo circuit in Figure 3 shows the *Xrouter* and other channels that are used to manage the coordination and facilitate the communication among rebecs.

For communication between rebecs, we need FIFO and filter channels. The message queues of rebecs are modeled by FIFO channels. We need to design a circuit to allow only the messages that are sent to a specific rebec to get into its queue, and filter out all other messages. To have an elegant design, we consider a consistent pattern of wiring between components. In Figure 3, there are fork nodes named F_i, and merge nodes named M_i. All messages that are sent by a rebec $rebec_i$ get out of its port *send*, then pass a *Sync* channel and enter the corresponding fork node F_i. Here, a message is copied into all the source channel ends of the outgoing *Sync* channels that are merged again in the node M_i. For a model with n rebecs, there are n *Sync* channels that connect each rebec to all other rebecs and carry the messages. Following each merge node M_i there is a *Filter* channel whose filter pattern is the ID of the receiver rebec. So, the filter following the node M_i filters out every message whose receiver is not $rebec_i$,

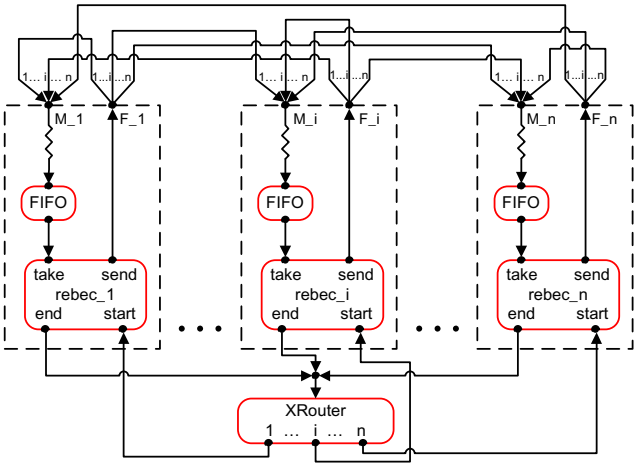

Fig. 3. Modeling Rebeca in Reo

allowing only the proper messages to pass through and get into the message queue of the rebec (the FIFO in Figure 3).

Upon receiving a *start* signal, a rebec takes a message from its queue by enabling the *take* port, and then executes the corresponding message server. During this execution, the messages that are sent, flow out of the rebec component through its *send* port, and arrive at the message queue of the destination rebec properly, passing the fork node, the merge node, and the filter channel.

Now, we have a Reo circuit that models a Rebeca model. But, to be able to construct the compositional semantics of a model and verify its properties we need to have a proper semantics for this Reo circuit and also for the rebecs. Constraint automata [19] are presented as a compositional semantics for Reo circuits and can be used to model components and the glue code circuit in a consistent way. They also provide verification facilities.

Looking more carefully, we see that by adding or removing rebecs the Reo circuit in Figure 3 which acts as the glue code will be changed. Our goal in obtaining the compositional semantics of the model is to be able to reuse the constraint automata of the parts of the Reo model that are not changed and not to construct the constraint automata of the whole Reo model from scratch. Observing that the glue code will change with a single change in the set of constituent rebecs, we can see that there is no gain in this way of constructing the compositional semantics. Although, the constraint automata for each rebec does not change and can be reused, all the join operations must be done again.

This is a good example to show how the modules in an object-based model are more tightly coupled than the modules in a component-based model. We changed our Reo circuit in Figure 3 to the circuit in Figure 4 to gain more modifiability and reusability. Here, the coordination part which is an *Xrouter* in Figure 3 is replaced with a compositional variant in Figure 4. Also, the communication part is changed to the simple circuit shown in Figure 4. This simplification is only

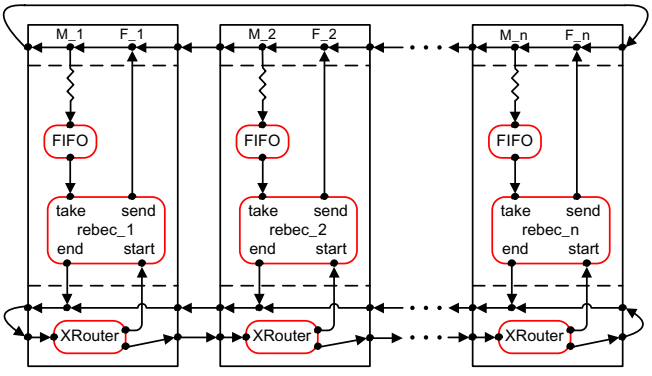

Fig. 4. Compositional modeling of Rebeca in Reo

valid because of the interleaved execution of each rebec and the fact that there is only one message carried through the *Sync* channels in each atomic step. In this way we have each rebec and its coordination and communication part as a component which can be plugged into or removed from a model without changing the rest of the model. Hence, by adding or removing a rebec, the entire model will not change. Note that our goal here is not to achieve exogenous coordination, because in Rebeca (like other object oriented models) the driving control and coordination are built in the code of rebecs and the message passing pattern.

5 Constraint Automata: Compositional Semantics of Reo

Constraint automata are presented in [19] to model Reo connectors. We use constraint automata to model the components, yielding Rebeca models fully as constraint automata. In this section, we explain the definition of constraint automata and how the constraint automata of a Reo circuit is constructed compositionally.

Using constraint automata as an operational model for Reo connectors, the automata-states stand for the possible configurations (e.g., the contents of the FIFO-channels of a Reo-connector) while the automata-transitions represent the possible data flow and its effect on these configurations. The operational semantics for Reo presented in [16] can be reformulated in terms of constraint automata. Constraint automaton of a given Reo connector can also be defined in a *compositional* way. For this, the composition operator for constraint automata and the constraint automata for a set of Reo connector primitives are presented in [19].

Definition 1. *[Constraint automata]* A constraint automaton (over the data domain *Data*) is a tuple $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ where

- Q is a set of states,
- $\mathcal{N}ames$ is a finite set of names,

- $\longrightarrow$ is a subset of $Q \times 2^{\mathcal{N}ames} \times DC \times Q$, called the transition relation of $\mathcal{A}$, where DC is the set of data constraints,
- $Q_0 \subseteq Q$ is the set of initial states.

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \longrightarrow$. We call N the name-set and g the guard of the transition. For every transition

$$q \xrightarrow{N,g} p$$

we require that (1) $N \neq \emptyset$ and (2) $g \in DC(N, Data)$. $\mathcal{A}$ is called finite iff Q, $\longrightarrow$ and the underlying data domain $Data$ are finite. □

Figure 5.a shows a constraint automaton for a 1-bounded FIFO channel with input port (source end) A and output port (sink end) B. Here, we assume that the data domain consists of two data items 0 and 1. Intuitively, the initial state q_0 stands for the configuration where the buffer is empty, while the states p_0 and p_1 represent the configurations where the buffer is filled with one or the other data item.

We now explain how constraint automata can be used to model the possible data flow of a given Reo circuit. The nodes of a Reo-circuit play the role of the ports in the constraint automata. To provide a *compositional* semantics for Reo circuits, we need constraint automata for all basic channel connectors and automata-operations to mimic the composition offered by the Reo-operations for join and hiding.

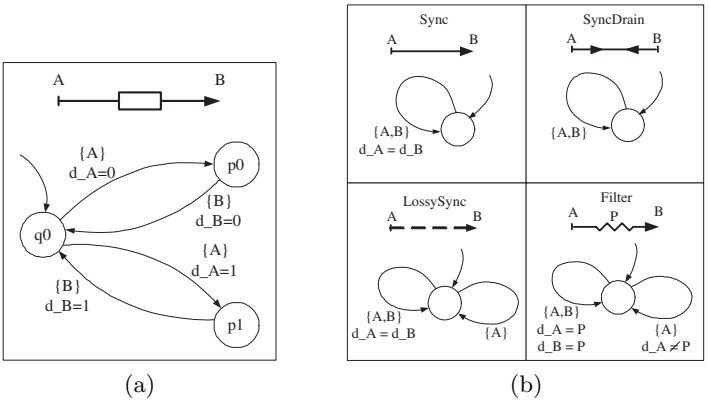

Fig. 5. (a) Deterministic constraint automaton for a 1-bounded FIFO channel; and, (b) Deterministic constraint automaton for some other channels

Figure 5.b shows the constraint automata for some of the standard basic channel types: a synchronous channel, a synchronous drain, a lossy synchronous channel, and a filter with pattern P. In every case, one single state is sufficient. Moreover, the automata are deterministic. There are operators defined on constraint automata that capture the meaning of Reo's join and hiding operators [19].

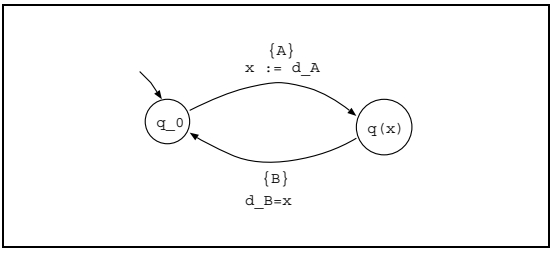

Fig. 6. Parameterized constraint automaton for a 1-bounded FIFO channel

Parameterized Constraint Automata. To simplify the pictures for constraint automata for data-dependent connectors, we use a parameterized notation for constraint automata, as proposed in [22]. For example, Figure 6 shows a parameterized constraint automata for a FIFO1 channel with source A and sink B. Thus, $q(x)$ in Figure 6 represents the states $q(d)$ for $d \in Data$. The transition from q_0 to $q(x)$ in the picture is a short-hand notation for the transitions from q_0 to $q(d)$ with the name-set $\{A\}$ and the data constraint $d = d_A$ where d ranges over all data elements in $Data$.

Formally, a *parameterized constraint automaton* is defined as a tuple

$$\mathcal{P} = (Loc, Var, v, \mathcal{N}ames, \rightsquigarrow, Loc_0, init)$$

where

- Loc is a set of locations,
- Var is a set of variables,
- $v : Loc \to 2^{Var}$ assigns to any location ℓ a (possibly empty) set of variables,
- $\mathcal{N}ames$ is a finite set of names (like in constraint automata),
- $\rightsquigarrow$ is a subset of $Loc \times 2^{\mathcal{N}ames} \times PDC \times X \times Loc$, called the transition relation of $\mathcal{P}$, where PDC is the set of parameterized data constraints and X is the function showing assignments to variables,
- $Loc_0 \subseteq Loc$ is a set of initial locations,
- $init$ is a function that assigns to any initial location $\ell \in Loc_0$ a condition for the variables.

$v(\ell)$ can be viewed as the parameter list of location ℓ. For instance, in Figure 6 we use $q(x)$ to denote that q is a location with parameter list $v(q) = \{x\}$, while q_0 is a location with an empty parameter list. The initial condition for q_0 is omitted which denotes that $init(q_0) = \mathsf{true}$.

6 Compositional Semantics of Rebeca Using Constraint Automata

To obtain the constraint automata of the coordination and communication parts of the Rebeca model, which are modeled in Reo, we use the join and hide operations on constraint automata. For specifying the semantics of rebecs we need

parameterized constraint automata. To obtain the parameterized constraint automaton (PCA) of each rebec, we use an algorithm, shown in Figure 7, to extract the PCA directly from the Rebeca code.

In the parameterized constraint automaton for each rebec i,

$$\mathcal{P}_i = (Loc_i, Var_i, v_i, \mathcal{N}ames_i, \rightsquigarrow_i, Loc_{0i}, init_i)$$

where we have $\mathcal{N}ames_i = \{start, end, send, take\}$, and $Loc_{0i} = \{idle\}$. For each rebec Var_i includes state variables of the rebec, local variables of each method, and *sender* variable which holds the ID of the sender of each message.

VARS: sender; {state variables}; {local variables};

BEGIN
Create locations: *Idle*, *Dispatch*
Create transitions:
$Idle \xrightarrow{\{start\}} Dispatch$
$Dispatch \xrightarrow[\text{sender} := \text{d}_{\text{take}}.\text{sender}]{\{\text{take, end}\},\ \text{d}_{\text{take}}.\text{msg=empty}} Idle$
FOR each message server $\mathcal{M}$ **DO**
Create transition: $Dispatch \xrightarrow[\text{sender} := \text{d}_{\text{take}}.\text{sender}]{\{\text{take}\},\ \text{d}_{\text{take}}.\text{msg}= \mathcal{M}} start_{\mathcal{M}}$
Create *control graph* from $start_{\mathcal{M}}$ to $end_{\mathcal{M}}$
Create transition: $end_{\mathcal{M}} \xrightarrow{\{end\}} Idle$
OD
END

Fig. 7. Algorithm to construct parameterized constraint automaton from a rebec code

The initial state of the PCA (Parameterized Constraint Automaton) of each rebec is denoted as the *idle* state. At the beginning all rebecs are in their *idle* states. By getting the *start* signal as input from the *Xrouter*, a rebec moves to its *Dispatch* state, where a message is taken from top of the corresponding queue. The data item of the port *take* is assumed to be a tuple consisting of the *sender* of the message and the *message server name*. According to the *d_take*, the next state is chosen. If the message queue is empty the transition goes back to the *idle* state. If not, the transition goes to the state which is the beginning of the execution of a message server. In fact, the second item of *d_take* which is the *message server name* specifies the next state. Suppose the message $\mathcal{M}$ is taken from the queue. This causes a transition to state $start_{\mathcal{M}}$, which denotes the beginning of the execution of the message server of $\mathcal{M}$.

The execution of each message server can be shown with a *control graph* representing its different branches and assignments. In this control graph, each send statement contributes to a transition. The name of this transition is *send*, and its data constraint is a tuple containing the name of the message being sent, the ID of the receiving rebec, and the ID of the sender which is *self*. In this phase, since the automata are created for the reactive classes and not the rebecs, the receiving rebec is chosen as one of the known rebecs. This ID is

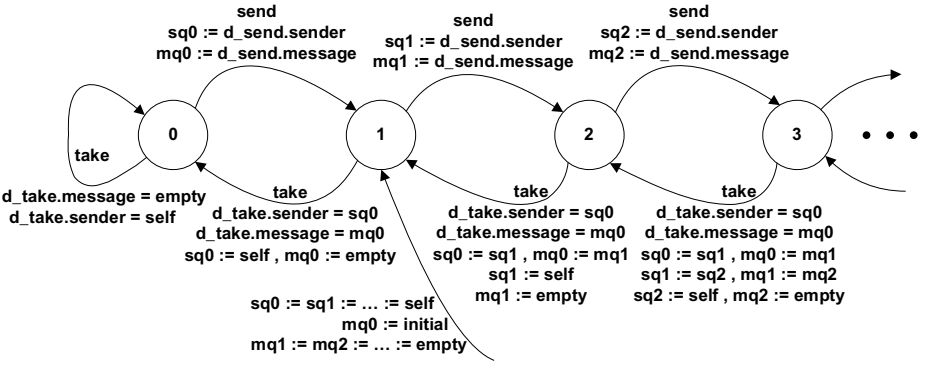

Fig. 8. Constraint automaton for a message queue channel

used by the designated *filter* of each rebec to identify the real receiver of the message. Each transition due to a *send* should also contain all the assignments made before that send. The assignments after the last send (if any) constitute the final transition of the control graph. This is a transition with *end* signal which connects the last state of the control graph ($end_{\mathcal{M}}$) to the *idle* state. This transition can be combined with the last transition of the control graph (and hence removing $end_{\mathcal{M}}$) to reduce the number of states. We use the bridge controller example of Section 2, to explain the algorithm in more detail in the next section.

We use a special kind of a FIFO channel to model the message queue of a rebec. The main point is that we want to be able to realize the situation when the queue is empty. This cannot be done with the conventional definition of a FIFO channel in Reo [17, 16]. We assume that there is a special data denoted by *empty* that the channel emits to show that the queue is empty. We define the behavior of the message queue channel as the constraint automaton shown in Figure 8.

7 An Example: Bridge Controller

We use a bridge controller as an example to model by constraint automata. This example is described in Section 2, and its Rebeca code is shown in Figure 1.

Figure 9.a shows the constraint automaton for the trains and Figure 9.b shows the constraint automaton for the bridge controller. The initial state for a train is the *idle* state. We move to the *Dispatch* state by receiving the *start* signal. A train has four message servers: *initial*, *YouMayPass*, *Passed*, and *ReachBridge*. For each one of these message servers there is an outgoing transition from the *Dispatch* state. Each transition goes to a state that designates the start of its corresponding message server. There is also another transition that is chosen when the message queue is empty. This one goes back to the *idle* state and outputs the *end* signal.

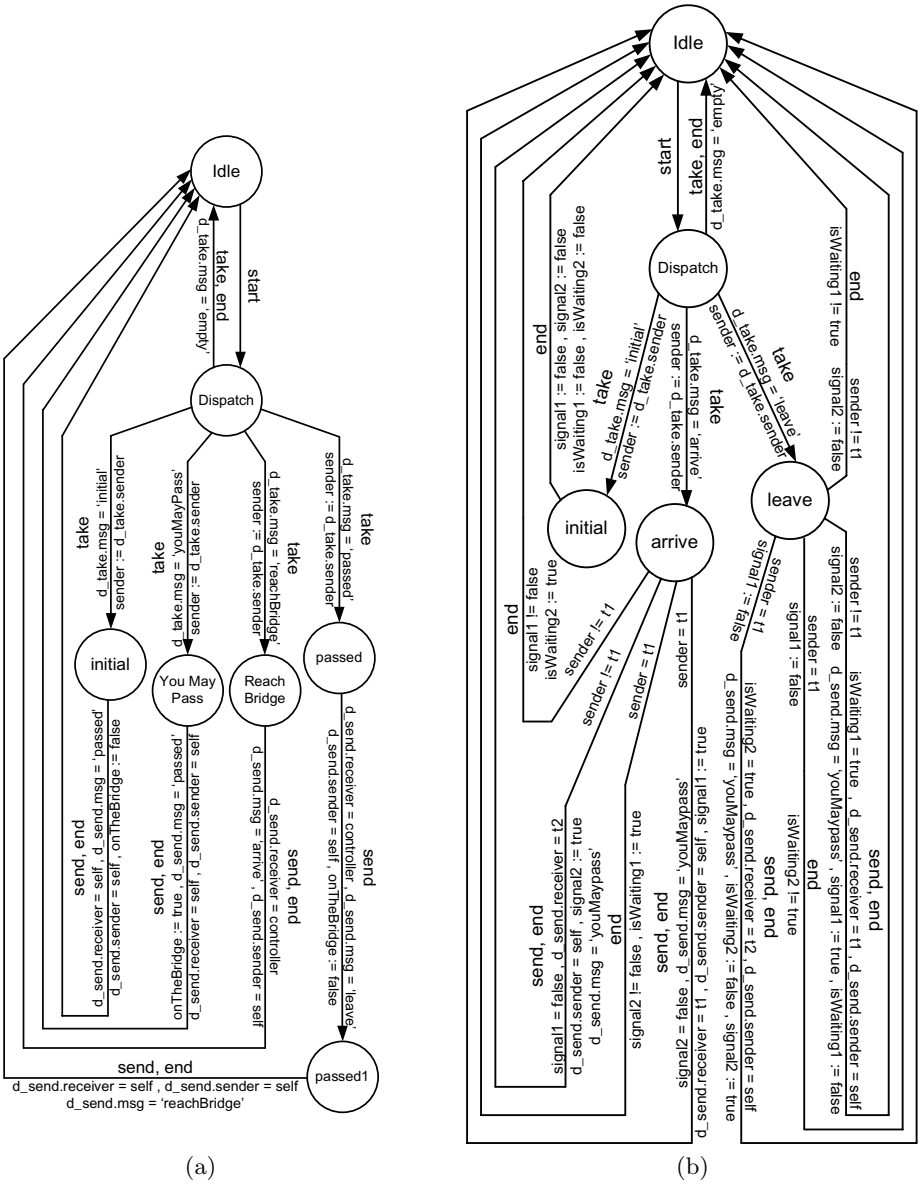

Fig. 9. Constraint Automata models for (a) Train; and, (b) Controller

As described in the algorithm of Figure 7, we must consider the different flows of control in each message server as a 'control graph'. In the message servers of the trains we have a single path in the flow of control. We partition each path by the *send* statements. For example in the message server *Passed* we have two fragments. We have two transitions corresponding to the *send* statements in Figure 9.a. The *end* signal is added to the last transition, which can be

considered as an optimization issue. Considering the controller, we have conditional statements in message servers *Arrive* and *Leave*, and hence more than one possible path in the flow of control. The transitions generated for different flows of controls can be seen in Figure 9.b.

The mapping presented in Section 4 and Figure 4 allow us to first construct the constraint automata of the communication and coordination parts, which can be reused in all Rebeca models. We can subsequently compose the constraint automata of the rebecs with these constraint automata. Thus, we obtain the constraint automaton of the whole system which shows the behavior of the model and can also be used for model checking purposes. We have already developed a tool to automate the specified mapping [23], and we have used it to map a few case studies in Rebeca into constraint automata. In this tool a set of heuristic rules are used to sequence a compositional construction of constraint automata that help to prevent the state space explosion problem.

Our compositional semantics allows a natural modular mapping from the problem space into the model space. To better show the benefit of this mapping, consider a modified version of the bridge controller problem, where more than one train can arrive from each side of the bridge (on multiple tracks). To avoid "hard-coding" the number of trains in this example, it is more appropriate to use a more component-based style model, where a queue on each side of the bridge keeps the passage requests. The Rebeca code for this version of bridge controller can be found on the Rebeca home page [24]. In this model, trains can be plugged in, and the derived constraint automata can be reused and composed together with the constraint automata of the new trains.

8 Conclusion and Future Work

We use the coordination language Reo to build a compositional semantics for the actor-based language, Rebeca. We modified the Reo circuit from its primary and natural layout to a more compositional and hence more reusable variant. Constraint automata are the essential devices in building this compositional semantics. The work presented in this paper can be used for both modeling and verification purposes. In general, for the object-based models that are written in a component-based paradigm, the compositional semantics presented here can be fully exploited and the unchanged parts can be completely reused. For all kinds of models, the constraint automata of the coordination and communication parts and the individual rebecs can be reused.

Our work can also be regarded as a good example where constraint automata are used for modeling components and connectors in a consistent manner, allowing to derive the behavior of a whole system as a composition of the behavior of its constituents. The differences between objects and components, and the challenges in moving from objects to components are illustrated in this work.

In our future work, we intend to use the tool in [23] for further experiments. We will continue our investigation of mapping reactive objects to components in order to characterize the patterns in the behavior of rebecs that make a model

more modifiable and the rebecs more reusable. Another direction in our future work is to consider dynamic rebec creation and dynamic changing topology in the mapping, although dynamic features are not yet supported by constraint automata they are present in Reo. A formal proof for our mapping algorithm will also be provided.

References

1. de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: Formal Methods for Components and Objects, International Symposium, FMCO'02, Leiden, The Netherlands, November 2002, Revised Lectures. Volume 2852 of LNCS, Springer-Verlag, Germany (2003)
2. de Roever, W.P., Langmaack, h., Pnueli, A., eds.: Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 1997, Revised Lectures. Volume 1536 of LNCS, Springer-Verlag, Germany (1998)
3. Lamport, L.: Composition: A way to make proofs harder. In: Proceedings of COMPOS: International Symposium on Compositionality: The Significant Difference. Volume 1536 of LNCS, (Springer-Verlag, Germany, 1997) 402–407
4. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, MIT (1987)
5. Abadi, M., Lamport, L.: Composing specifications. In Jagadish, H.V., Mumick, I.S., eds.: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada, ACM Press, USA, (1996) 365–376
6. Talcott, C.: Composable semantic models for actor theories. Higher-Order and Symbolic Computation **11** (1998) 281–343
7. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informatica **63** (Dec. 2004) 385–410
8. Sirjani, M., Movaghar, A.: An actor-based model for formal modelling of reactive systems: Rebeca. Technical Report CS-TR-80-01, Tehran, Iran (2001)
9. Sirjani, M., Movaghar, A., Mousavi, M.: Compositional verification of an object-based reactive system. In: Proceedings of AVoCS'01, Oxford, UK (2001) 114–118
10. Hewitt, C.: Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT (1972)
11. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1990)
12. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Model checking, automated abstraction, and compositional verification of Rebeca models. Journal of Universal Computer Science **11** (2005) 1054–1082
13. Sirjani, M., Shali, A., Jaghoori, M., Iravanchi, H., Movaghar, A.: A front-end tool for automated abstraction and modular verification of actor-based models. In: Proceedings of ACSD'04, (IEEE Computer Society, 2004) 145–148
14. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Movaghar, A.: Efficient symmetry reduction for an actor-based model. In: 2nd International Conference on Distributed Computing and Internet Technology. Volume 3816 of LNCS. (2005) 494–507
15. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: The model-checking engine of Rebeca. In: ACM Symposium on Applied Computing - Software Verificatin Track. (2006) to appear.

16. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science **14** (2004) 329–366
17. Arbab, F., Rutten, J.J.: A coinductive calculus of component connectors. Technical Report SEN-R0216, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands (2002)
18. Arbab, F.: Abstract behavior types: A foundation model for components and their composition. In: Proceedings of FMCO'03. Volume 2852 of LNCS. (2003) 33–70
19. Arbab, F., Baier, C., Rutten, J.J., Sirjani, M.: Modeling component connectors in Reo by constraint automata. In: Proceedings of FOCLASA'03. Volume 97 of ENTCS., Elsevier (2004) 25–46
20. Sirjani, M., de Boer, F.S., Movaghar, A., Shali, A.: Extended Rebeca: A component-based actor language with synchronous message passing. In: Proceedings of ACSD'05, IEEE Computer Society (2005) 212–221
21. Sirjani, M., de Boer, F.S., Movaghar, A.: Modular verification of a component-based actor language. Journal of Universal Computer Science **11** (2005) 1695–1717
22. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.: Modeling component connectors in Reo by constraint automata. (Science of Computer Programming) accepted 2005, to appear.
23. Farrokhian, M.: Automating the mapping of Rebeca to constraint automata. Master Thesis, Sharif University of Technology (2006)
24. Rebeca home page: Available through http://khorshid.ut.ac.ir/∼rebeca.