Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 130 (2005) 345–377
www.elsevier.com/locate/entcs

A Rewriting Semantics for a Software Architecture Description Language

Alexandre Rademaker [2]

Instituto de Computação
Universidade Federal Fluminense
Niterói, Brazil

Christiano Braga [3]

Instituto de Computação
Universidade Federal Fluminense
Niterói, Brazil

Alexandre Sztajnberg [1]

Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro
Rio de Janeiro, Brazil

Abstract

Distributed and concurrent application invariably have coordination requirements. The design of those applications, composed by several (possibly distributed) components, has to consider coordination requirements comprising inter-component interaction styles, and intra-component concurrency and synchronization aspects. In our approach coordination aspects are treated in the software architecture level and can be specified in high-level contracts in CBabel ADL. A rewriting logic semantics for the software architecture description language CBabel is given, revisiting and extending previous work by some of the authors, which now includes a revision of the previous semantics and the addition of new features covering all the language. The CBabel tool is also presented. The CBabel tool is a prototype executable environment for CBabel, that implements the given CBabel's rewriting logic semantics and allows the execution and verification of CBabel descriptions in the Maude system, an implementation of rewriting logic. In this way, software architectures describing complex applications can be formally verified regarding properties such as deadlock and synchronization consistency in the software architecture design phase of its life cycle.

Keywords: software architecture description languages, rewriting logic, Maude, contracts

doi:10.1016/j.entcs.2005.03.018

1 Introduction

Distributed and concurrent applications invariably have coordination requirements. The design of those applications, composed by several (possibly distributed) components, has to consider coordination requirements comprising inter-component interaction styles, and intra-component concurrency and synchronization aspects. For instance, components can interact using synchronous or asynchronous method invocation. Also, invocation of two or more methods might have to be constrained regarding concurrency and synchronization. Coordination requirements are usually attended in the programming phase by using language constructions [19,17] or library services [3,2] provided with the operating system. This often leads to *ad hoc* approaches, were the coordination code is tangled with the functional code, resulting in less reusable modules and error prone solutions [14,25]. Even when using contemporary techniques such as design patterns, it is up to the designer using the recurrent designs [34] with separation of concerns and modularity in mind.

In the CR–RIO framework [21,37], which is based on meta-level and architecture configuration approaches, the coordination aspects can be treated in the software architecture level using the CBabel ADL. The purpose of using software architecture description languages (ADLs) is to keep separated the description of how distributed components are *connected* from the descriptions of the *internal behavior* of each component. The separation-of-concerns provided by architectural descriptions has several interesting properties including modularity of the architectural descriptions, reuse of components in different architectures, and (dynamic) reconfiguration of architectures.

CBabel is an ADL that, besides the usual architectural primitives [35] such as components and ports, provides contracts [33,16,4] as first class constructions. In that way, coordination aspects can be described with CBabel contracts. Basically, the designer can describe mutual exclusion properties constraining the use of input ports, or in-ports for short, of a functional module and specify guards to govern synchronization and consistency properties for those in-ports. The described coordination aspects are encapsulated in connectors that mediate all interactions among functional modules. With this approach, we separate coordination aspects concerns from functional aspects, which do not need to be included in the design or implementation of functional modules. In fact different instances of a same component can be submitted to distinct coordination specifications.

[1] Email: alexszt@ime.uerj.br
[2] Email: arademaker@ic.uff.br
[3] Email: cbraga@ic.uff.br

In [5] the basic ideas of a rewriting logic [28] semantics for CBabel [21] were presented. In the present paper we give the *complete* formalization of CBabel in rewriting logic, that is, its rewriting semantics, and *also* present the *CBabel tool*, an executable environment for CBabel implemented in the Maude system [8]. CBabel tool is a direct implementation of the rewriting semantics of CBabel which allows the execution and verification of CBabel descriptions. We focus on the formalization of CBabel and the use of the CBabel tool. A detailed report on the implementation of the CBabel tool will be given elsewhere.

Rewriting logic is a logic and semantic framework to which several models of computation, logics and specification languages have been mapped to [24], due to its unified view of computation and logic. Maude is a high-performance implementation of rewriting logic and a powerful *meta-tool* when its meta-programming facilities are joined with the analysis tools available either built-in in the system, such as the LTL model checker [12], or developed in Maude itself, such as Clavel's Inductive Theorem Prover [7]. Maude has also an extensible *module algebra* implemented in Full Maude [11], which also endows the Maude system with an object-oriented syntax.

CBabel tool is implemented precisely as a *conservative extension* of Full Maude, following a very natural interpretation of CBabel concepts in object-oriented terms. Roughly, components are represented by classes, component's instances as objects, port declarations as messages and port stimulus as message passing [5]. The rewriting semantics that we have given to CBabel uses the object-oriented notation for rewriting logic and is implemented as a transformation function in Maude using Maude's meta-programming capabilities. This transformation function is the core of the CBabel tool execution environment prototype.

The remaining of the paper is organized as follows. Section 2 exemplifies the CBabel syntax. Section 3 gives the necessary background in rewriting logic and object-oriented rewrite theories in Maude. Section 4 defines the rewriting semantics of CBabel. Section 5 presents the execution and verification of CBabel descriptions in Maude using the CBabel tool. In Section 6 we discuss related work. Section 7 concludes this paper with our final remarks.

2 CBabel Descriptions

In this section we introduce the syntax of CBabel by means of three variants of the classic producer-consumer-buffer example. Later in Section 5 we will verify these specifications in Maude using the CBabel tool. It is worth emphasizing that the Maude specifications are *automatically* generated by CBabel tool, as

opposed to our previous work in [5].

In the producer-consumer-buffer example there is a producer willing to access a buffer, which may be bounded, to add an item it has just produced, and a consumer willing to access the buffer to consume an item from the buffer. There are at least two problems in such a situation: (i) the producer and the consumer should not access the buffer at the same time, which is the so called race condition, and (ii) the buffer is bounded and the producer should not add more items than the buffer can hold and the consumer should not remove an item from an empty buffer. (Actually, in Section 5, we also check for *deadlocking*.)

The first CBabel architecture that specifies the producer-consumer-buffer, and the respective informal graphic representation, are given in Figure 1. Please observe that in the graphic representation the name of the components and their respective input and output ports are represented, to help the reader following the CBabel code. Modules specify the component's interfaces that will be used in an architecture configuration (`PRODUCER`, `CONSUMER` and `BUFFER`, in this example). A special module, called an *application*, declares how each component should be instantiated and how they should be linked together. In Figure 1 the application module is named `PC-DEFAULT`. It creates one instance for each component and link them together through their *ports*. There are input and output ports. Input ports may be informally understood as the "services" that a component *provides* and output ports as the services that a component *requires*. Therefore, in our example, a producer needs a service to deliver or put an item, and respectively, a consumer to get. A buffer module offers services to put items in and get items from its internal storage. The actual request of a service occurs through *port stimuli*, that is, the fact that a producer is requesting to put an item is represented by a stimulus to its put port. In the same way, a buffer offering or providing its service to put (resp. get) an item to (resp. from) its internal buffer is represented by a stimulus to its put (resp. get) port.[4] A sequence of such port stimuli is called an *interaction*. Ports may communicate *asynchronously* and *synchronously*. In the latter case output ports expect a returning or acknowledging stimulus from the input port linked to it, which is not true in the former case. Asynchronous ports are specified using the `oneway` keyword. The examples in Figures 1, 2, and 3 declare synchronous ports.

As we mentioned before, this architecture has both a race condition prob-

[4] One may have noticed that the actual *item* type is not declared as a parameter of `producer@put`, for instance. This is due to the fact that we are actually interested in verifying properties related to the control flow of messages in an architecture and not in the properties about the data being carried by the messages.

```
module PRODUCER {
 out port producer@put ;
}

module CONSUMER {
 out port consumer@get ;
}

connector DEFAULT {
 in port default@in ;
 out port default@out ;
 interaction{
   default@in > default@out ;
 }
}
```

```
application PC-DEFAULT {
 instantiate BUFFER as buff ;
 instantiate PRODUCER as prod ;
 instantiate CONSUMER as cons ;
 instantiate DEFAULT as default1 ;
 instantiate DEFAULT as default2 ;
 link prod.producer@put to default1.default@in ;
 link default1.default@out to buff.buffer@put ;
 link cons.consumer@get to default2.default@in ;
 link default2.default@out to buff.buffer@get ;
}

module BUFFER {
 var int buffer@items = int(0) ;
 var int buffer@maxitems = int(2) ;
 in port buffer@put ;
 in port buffer@get ;
}
```

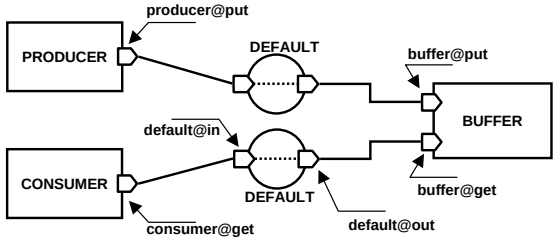

Fig. 1. Producer-consumer-buffer basic architecture

lem between `prod` and `cons`, instances of `PRODUCER` and `CONSUMER`, respectively, and also may have overflow and underflow problems if the buffer is bounded. To solve the race condition problem one could use a mutual exclusive contract to coordinate the access of the producer and the consumer to the buffer, since an interaction will occur either through port `mutex@in1` or `mutex@in2`. Figure 2 presents a new application module that connects a producer, a consumer and a buffer through a mutual exclusion connector that mediates the access to the buffer.

The problem of bounded access to the buffer still exists in the architecture of Figure 2. To solve this problem one may use a *guarded interaction contract* or simply guard contract. It specifies that two ports may interact if a certain condition holds. Once the condition holds the *before* block of the contract is executed. When the acknowledge stimulus arrives to the output port, the *after* block of the guard contract is executed. Another concept that is typically used together with guard contracts is *state* variables. State variables are shared-memory variables. A change to one such variable by one component is immediately noticed by another component that is bound to it. In CBabel state variables are declared in the components that share the variables and the application module *binds* them together specifying that when one of them

```
connector MUTEX {
 in port mutex@in1 ;
 in port mutex@in2 ;
 out port mutex@out1 ;
 out port mutex@out2 ;
 exclusive{
  mutex@in1 > mutex@out1 ;
  mutex@in2 > mutex@out2 ;
 }
}
```

```
application PC-MUTEX {
 instantiate BUFFER as buff ;
 instantiate PRODUCER as prod ;
 instantiate CONSUMER as cons ;
 instantiate MUTEX as mutx ;
 link prod.producer@put to mutx.mutex@in1 ;
 link mutx.mutex@out1 to buff.buffer@put ;
 link cons.consumer@get to mutx.mutex@in2 ;
 link mutx.mutex@out2 to buff.buffer@get ;
}
```

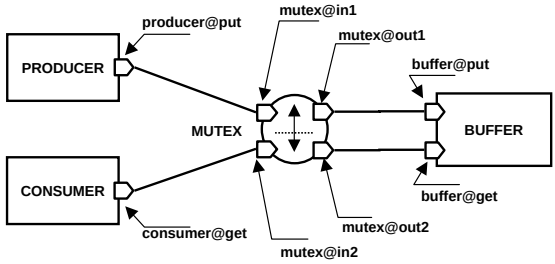

Fig. 2. MUTEX connector and the PC-MUTEX application

has changed the other should immediately notice the change.

Figure 3 specifies two guards in the connectors GUARD-PUT and GUARD-GET, to control the access to the buffer from the producer and from the consumer, respectively. They control the access to the buffer by inspecting selected state variables, to either forward or block the interaction flow. The variable gp@nEmpty is bound to gg@nEmpty, and the variable gg@nFull is bound to gp@nFull as described in PC-GUARDS-MUTEX application module. Whenever gp@nEmpty, number of empty spaces in the buffer, becomes zero, GUARD-PUT's guard blocks any insertion into the buffer. The opposite happens when the buffer has no items at all, gg@nFull equal zero, and then GUARD-GET's guard blocks any removal from the buffer. An important observation is that in the guard contracts local variables are used to drive the synchronization mechanism. In the BUFFER module there are others local variables that control the actual state of the buffer. This helps to further separate coordination concerns form the buffer implementation. Notice that in the graphic representation only the ports of the connectors added to this new configuration, GUARD-PUT and GUARD-GET, are depicted. Also notice the dotted double-arrow between these connectors, representing the binding of the local variables described in the code.

Coordination contracts specified in the architectural level such as the one just described can be used to generate deployable code [37] and allows employing formal techniques to verify several properties of the intended design before stepping to an implementation phase [23]. In Section 4 we present our approach to that challenge.

```
connector GUARD-PUT {
 var int gp@nFull = int(0) ;
 staterequired int gp@nEmpty ;
 in port gp@in ;
 out port gp@out ;

 interaction {
  gp@in >
  guard(gp@nEmpty > int(0)) {
   before {
    gp@nEmpty = gp@nEmpty - int(1) ;
   }
   after {
    gp@nFull = gp@nFull + int(1) ;
   }
  } > gp@out ;
 }
}
```

```
connector GUARD-GET {
 var int gg@nEmpty = int(2) ;
 staterequired int gg@nFull ;
 in port gg@in ;
 out port gg@out ;

 interaction {
  gg@in >
  guard(gg@nFull > int(0)) {
   before {
    gg@nFull = gg@nFull - int(1) ;
   }
   after {
    gg@nEmpty = gg@nEmpty + int(1) ;
  }
   } > gg@out ;
  }
 }
```

```
application PC-GUARDS-MUTEX {
 instantiate BUFFER as buff ;
 instantiate PRODUCER as prod ;
 instantiate CONSUMER as cons ;
 instantiate MUTEX as mutx ;
 instantiate GUARD-GET as gget ;
 instantiate GUARD-PUT as gput ;

 link prod.producer@put to gput.gp@in ;
 link cons.consumer@get to gget.gg@in ;
 link gput.gp@out to mutx.mutex@in1 ;
 link gget.gg@out to mutx.mutex@in2 ;
 link mutx.mutex@out1 to buff.buffer@put ;
 link mutx.mutex@out2 to buff.buffer@get ;

 bind int gget.gg@nFull to gput.gp@nFull ;
 bind int gput.gp@nEmpty to gget.gg@nEmpty ;
}
```

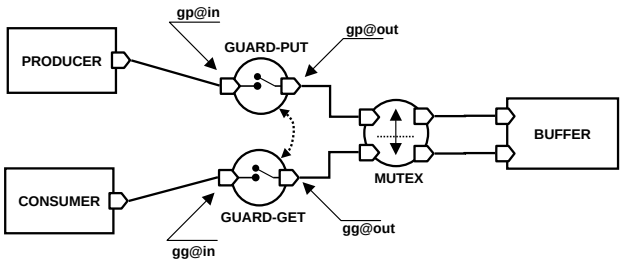

Fig. 3. Guarded connectors and the PC-GUARDS-MUTEX application

3 Rewriting Logic and Maude

A rewrite theory R is a tuple (Σ, E, R), where Σ is the rewrite theory's signature, E is the set of equations and R is the set of rewrite rules. The set E of equations is assumed *confluent* and *terminating*, which roughly means that every term should have a unique normal form and that should be no infi-

nite chain of equational simplification steps. The rules R should be *coherent*, that is, alternating between equations and rules does not lose rewrite computations. Rules are applied *modulo* E, that is, the rewrite relation is defined on the equivalence classes of terms in the initial algebra of the equational specification (Σ, E) with variables, $T_\Sigma(X)$.

Rewriting logic is parameterized by its underlying equational logic. (In particular membership equational logic [30], a generalization of order-sorted equational logic, is chosen in the Maude system.) Moreover, the notion of *frozen* operators [6] has been added to rewrite theories, generalizing them. However, to keep the presentation simple, in the following rules of deduction for rewriting logic we choose order-sorted equational logic as underlying logic and the version of rewriting logic where frozen operators are not considered.

- **Reflexivity**. For each term t in the initial algebra of Σ with variables $T_\Sigma(X)$,

$$\frac{}{(\forall X)t \longrightarrow t}$$

- **Equality**.

$$\frac{(\forall X)u \longrightarrow v \quad E \vdash (\forall X)u = u' \quad E \vdash (\forall X)v = v'}{(\forall X)u' \longrightarrow v'}$$

- **Congruence**. For each $f : k_1 \dots\ k_n \to k$ in Σ, with $t_i, t_i' \in T_\Sigma(X)_{k_i}, 1 \leq i \leq n$,

$$\frac{(\forall X)t_1 \longrightarrow t_1' \dots\ (\forall X)t_m \longrightarrow t_m'}{(\forall X)f(t_1, \dots\ t_m) \longrightarrow (\forall X)f(t_1', \dots\ t_m')}$$

- **Replacement**. For each finite substitution $\theta : X \to T_\Sigma(Y)$, and for each rule of the form,
  $l : (\forall X)t \longrightarrow t' \Leftarrow (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j w_j \longrightarrow w_j')$

$$\frac{\bigwedge_i (\forall Y)\theta(u_i) = \theta(u_i') \wedge \bigwedge_j (\forall Y)\theta(w_j) \longrightarrow \theta(w_j')}{(\forall Y)\theta(t) \longrightarrow \theta(t')}$$

- **Transitivity**.

$$\frac{(\forall X)t_1 \longrightarrow t_2 \dots\ (\forall X)t_2 \longrightarrow t_3}{(\forall X)t_1 \longrightarrow t_3}$$

Rewriting logic is a computational logic to specify concurrent systems. The inference rules above allows us to infer all the possible finitary concurrent

computations of a system specified as a rewrite theory as follows: (i) reflexivity is the possibility of having idle transitions, (ii) equality means that states are equal modulo E, (iii) congruence is a general form of sideways parallelism, (iv) replacement combines an atomic transition at the top using a rule with nested concurrency in the substitution, and (v) transitivity is sequential composition.

An important class of concurrent systems is that of concurrent object systems. Rewriting logic has an *object-based* notation, that was quite useful to us while giving the semantics for CBabel, since it is very natural to think of CBabel primitives in object-oriented terms, as we have already mentioned in Section 1.

In particular, object-oriented syntax in the Maude language [29] represents the concurrent state, or the system configuration, as a *multiset* of objects and messages, declared as juxtaposition with the following operator declaration.

```
op _ : Configuration Configuration → Configuration
                                    [ctor assoc comm id: null] .
```

The keyword `op` is used to declare an operator in Maude. The keywords `ctor`, `assoc`, `comm`, and `id` are attributes of the juxtaposition operator meaning that it is a constructor that satisfies the structural laws of associativity and commutativity and has identity `null`, declared as a constant operator of sort `Configuration`. Objects and messages are singleton configurations being subsorts of the configuration sort so that more complex configurations are generated out of them by multiset union. An object is represented as a term

$$\langle\ O : C \mid a_1 : v_1\ ,\ \ldots\ ,\ a_n : v_n \rangle$$

where O is the object's identifier, C is the object's class identifier, a_i's are the object's attributes, and v_i's are their corresponding values. The order of the attributes does not matter, so the `_,_` operator is also declared associative and commutative by means of the attributes `assoc` and `comm`. Classes are declared in Maude with syntax

$$\texttt{class}\ C \mid a_1 : s_1\ ,\ \ldots\ ,\ a_n : s_n\ .$$

where C is the class name and s_i is the sort required for attribute a_i. It is also possible to give subclass declarations, so that all attributes and rewrite rules of a superclass are inherited by a subclass which can have additional attributes and rules of its own. The syntax of messages is declared using the `msg` keyword in a way similar to an operator declaration. For instance, a message named `to` that is parameterized by the object identifier of the sender object, the object identifier of the receiver object and some data can be declared as:

```
msg to : Oid Oid Data → Msg .
```

The associativity and commutativity of configuration's multiset constructor

make it as a "soup" of objects and messages [32], so that any objects and messages can at any time come together and participate in a concurrent transition corresponding to a communication event. The concurrent interactions between objects are axiomatized by rewrite rules. The general form of such a rule is given in Maude as follows:

$$\begin{aligned}
&\texttt{crl}\ [r] : M_1\ \ldots\ M_n\ \langle\ O_1 : F_1 \mid atts_1\ \rangle\ \ldots\ \langle\ O_m : F_m \mid atts_m\ \rangle \Rightarrow \\
&\qquad \langle\ O_{i_1} : F'_{i_1} \mid atts'_{i_1}\ \rangle\ \ldots\ \langle\ O_{i_k} : F'_{i_k} \mid atts'_{i_k}\ \rangle \\
&\qquad \langle\ Q_1 : D_1 \mid atts''_1\ \rangle\ \ldots\ \langle\ Q_p : D_p \mid atts''_p\ \rangle \\
&\qquad M'_1\ \ldots\ M'_q \\
&\quad \texttt{if}\ \ C\ .
\end{aligned}$$

where r is the rule label, the Ms are message expressions, $i_1, \ldots, i_k$, are different numbers among the original $1, \ldots, m$, and C is the rule's condition.

4 An Object-Oriented Rewriting Semantics for CBabel Software Architecture Primitives

The fundamental software architecture elements of CBabel could be informally defined as follows. A *component* can be either a module or a connector. A module is a "wrapper" to an entity that performs a computation, such as an object or a function. A connector mediates the interaction among modules, governing how they communicate and coordinate. It is through a *port* that components communicate requesting functionalities or "services" from each other. Ports communicate following a message passing model. *Coordination contracts* define how a group of ports should interact. It may be sequentially, mutually exclusive, or guarded by a condition. An *application* is a special module that declares how each component should be instantiated, how components should be linked, and how state variables should be bound to each other. *Links* establish the connection of two ports enabling them to interact. *State required variables* allow for components to exchange information atomically, that is, within a shared-memory model of communication.

The following sections formalize these notions according to the mapping from CBabel elements to object-oriented concepts according to Table 1. In what follows, we have used the convention of writing small letters to represent elements of a set and capitalized letters to represent the sets themselves.

component	→	object-oriented theory
component instance	→	object
application state	→	multiset of objects
port	→	message declaration
port stimulus	→	message passing (rewrite rules)
link	→	unconditional equation
coordination contract	→	rewrite rules
local or state required variable	→	class attribute
bind of variables	→	equations

Table 1
Mapping from CBabel concepts to rewriting logic

4.1 Components

A component is either a module or a connector. A module may declare local variables, input ports and output ports. A connector may, in addition to the same declarations that may be done in a module, declare a coordination contract and state required variables.

Components are mapped to rewrite theories in rewriting logic. Each component gives rise to a class declaration in the associated rewrite theory's signature, named after the component's name, with a constructor method. A component instance is represented by an object instance of such class. Objects that represent modules may answer to messages *do* and *done*. These messages represent, or signalize, the beginning and end of a module's observable *internal* behavior. (This is important regarding verification issues as we shall see in Section 4.) Connectors, by their turn, react to messages guided by the described coordination contract. The messages carry the sequence of object identifiers in a given *interaction*, that is, a finite sequence of port stimuli of ports that are related by link declarations. The interaction sequence is necessary so that a component instance may be properly acknowledged in a synchronous interaction when there is more than one component instance linked to a given input port. Local variables in a CBabel module are mapped to class attributes in the associated class in rewrite theory's signature.

Let us formalize components in rewriting logic. Note, however, that the declaration of ports will be formalized in Section 4.2 and the formalization of coordination contracts is given in Section 4.3. A CBabel module declaration M is a tuple (n, V, I, O) where n is an identifier representing the module's

name, the set V of variable declarations holds triples (v, l, t) where v is an identifier representing the variable's name, l is the value which v should be initialized to, and t is the variable's type, which must be one of CBabel's built-in primitive types. Sets I and O are both sets of identifiers holding input and output port declarations respectively.

The concept of an *interaction*, informally described above, is formalized as a *stack* of pairs with the first projection being an object identifier and the second projection a port identifier, declared in the rewrite theory *CBABEL-CONFIGURATION*, which contains basic declarations that will be made explicit in the forthcoming sections, together with the declaration of messages

$$do\ ,\ done\ :\ Oid\ PortId\ Interaction \rightarrow Msg$$

The rewriting semantics of a CBabel module n is given by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ whose signature Σ imports the declarations of the *CBABEL-CONFIGURATION* rewrite theory, and a class declaration $class\ n \mid S$, where S is the attribute set of class n, whose elements are named after the elements of V. The signature Σ also includes the class constructor operator declaration $instantiate\text{-}n\ :\ Oid \rightarrow Object$. The set E of equations includes

$$eq\ instantiate\text{-}n(\omega)\ = < \omega\ :\ n \mid a_1\ :\ l_1,\ \ldots\ , a_n\ :\ l_n > \ . \tag{1}$$

where ω is an object identifier, each a_i is an object attribute named after v_i, and l_i is the value that initializes v_i.

Equation 1 specifies that given an object identifier, *instantiate-n* produces an object instance of class n with attributes initialized to the values l declared for the CBabel component variables in V.

4.2 Ports

A CBabel component may have input ports and output ports. Input ports are used to provide a service of a given component and output ports are used by a component to request a service from other components. Moreover, port communication may be synchronous or asynchronous. The latter case is declared in CBabel by means of the keyword `oneway`. The absence of the `oneway` keyword as a port declaration modifier means that communication through that port should be synchronous.

In a given CBabel component, port declarations are mapped to message declarations in the associated rewrite theory's signature. Port stimulus is represented, of course, as passing a message to the appropriate object, that is, to the object that represents the component linked to that port. (See Section 4.4 for the formalization of CBabel's link declaration.) However, instead of declaring one message for every port, we have chosen to declare two general messages *send* and *ack*, since it significantly simplifies the semantics. The

ports are then mapped to constants which parameterize these general messages. Messages *send* and *ack*, like *do* and *done*, carry the sequence of object identifiers in a given interaction.

The declaration of ports also includes rules in the associated rewrite theory. However, the treatment for rule generation is different for modules and connectors since connectors, as opposed to modules, declare contracts that coordinate the interactions, that is, the message flow among the objects that represent an architecture instance, also know as a topology. In the reminder of this section we will explain how rules are derived from port declarations in modules and Section 4.3 will give a detailed explanation on how rules are derived from port and coordination contracts declarations in a connector.

There are four different port declaration possibilities in modules which arise from combining synchronous and asynchronous communication with input and output port interaction.

- When a *synchronous input* port is declared in a CBabel component, two rules must be created: (i) one specifying that sending a message to that port should trigger an internal behavior to that component and (ii) another specifying that once that internal behavior is finished, an acknowledgment message should be sent back to the component that stimulated that port.

  Triggering a component's internal behavior is represented by a component sending a message *do* to itself. Once a component has finished performing its internal behavior it sends a message *done* to itself which is then turned into an acknowledgment message.
- When an *asynchronous input* port is declared, one rule should be added to the rewrite theory's rule set specifying that sending a message to that port should trigger that component's internal behavior.
- Declaring a *synchronous output* port should add two rules to the associated rewrite theory's rule set. The first rule specifies that when a component is *doing* one of its internal behaviors, a "service" from another component may be requested through that port. Moreover, this request should *block* that port until an answer to that request arrives, thus unlocking that port, which is specified by a second rewrite rule. The execution of that internal behavior is then considered *done*. The effect of locking and unlocking a port is captured by updating the status attribute *for that port* in the object that represents the CBabel component instance holding that port.
- The declaration of an *asynchronous output* port adds a rule to the rule set of the associated rewrite theory. The rule specifies that once that port is stimulated the associated message can be unconditionally rewritten since asynchronous ports do not require acknowledgment messages and therefore

do not need the treatment we have described for synchronous output ports in the previous bullet.

Let us now formalize this prose. First note that the mapping from a CBabel *component* port declaration to the associated rewrite theory signature is the same for both modules and connectors. However a different treatment is required for specifying *behavior*. As in our informal explanation above, in the remainder of this section we will formalize how the rewrite theory signature is affected by a port declaration in the associated CBabel component, and how the rule set of the rewrite theory is affected by port declarations in the associated CBabel module. The formalization of how rules are generated from port declarations and coordination contracts in a CBabel connector will be given in Section 4.3.

Given a CBabel module declaration (n, V, I, O) or a connector declaration (n, V, I, O, c) where n is the component's name identifier, V the variable declaration set, I the input ports declaration set, O the output port declaration set, and c the coordination contract declaration. The signature Σ of the rewrite theory associated to the CBabel component includes: (i) for each port declaration p in I, a constant p of sort *PortInId*, (ii) for each port declaration p in O, a constant p of sort *PortOutId*. The sorts *PortInId* and *PortOutId* are subsorts of *PortId*, the sort that parameterizes the generic messages $send, ack \ : \ Oid\ PortId\ Interaction \rightarrow Msg$. The sorts *PortId*, *PortInId*, and *PortOutId*, together with messages *send* and *ack* are declared in the rewrite theory *CBABEL-CONFIGURATION*, included in Σ.

The formalization of how port declarations in a CBabel *module* gives rise to rules in the associated rewrite theory. One should consider the four possible combinations for port declarations informally given above. Given a CBabel module declaration (n, V, I, O):

- The declaration of a *synchronous input* port i in I gives rise to Rules 2 and 3 in the associated rewrite theory rule set R:

  (2) $$rl\ send(\omega, i, \iota)\ < \omega\ :\ n \mid A > \Rightarrow\ do(\omega, i, \iota)\ < \omega\ :\ n \mid A >\ .$$

  (3) $$rl\ done(\omega, i, \iota)\ < \omega\ :\ n \mid A > \Rightarrow\ ack(\iota) < \omega\ :\ n \mid A >\ .$$

  where ω is the object identifier of the object that represents an instance of the CBabel module, ι is the interaction, and A is the object's attribute set.
- The declaration of an *asynchronous input* port gives rise to Rules 2 and 4:

  (4) $$rl\ done(\omega, i, \iota)\ < \omega\ :\ n \mid A > \Rightarrow < \omega\ :\ n \mid A >\ .$$
- The declaration of a *synchronous output* port $o \in O$ gives rise to Rules 5 and 6,

  (5) $$rl\ do(\omega, o, none)\ < \omega\ :\ n \mid o\text{-}status\ :\ unlocked, A > \Rightarrow$$

$$send(\omega, o, [\omega, o]) \ < \omega \ : \ n \mid o\text{-}status \ : \ locked, A > .$$

(6) $$rl \ ack([\omega, o]) \ < \omega \ : n \mid o\text{-}status : s, A > \Rightarrow$$
$$< \omega \ : \ n \mid o\text{-}status \ : \ unlocked, A > done(\omega, o, none) \ .$$

where s is a variable of sort *PortStatus*, declared in the rewrite theory *C-BABEL-CONFIGURATION* together with constants

$$locked \ , \ unlocked \ : \rightarrow \ PortStatus$$

, and *none* is the unit of *Interaction*.

- The declaration of an *asynchronous output* port gives rise to Rule 7:

(7) $$rl \ do(\omega, o, none) \ < \omega \ : \ n \mid A > \Rightarrow$$
$$send(\omega, o, [\omega, o]) \ < \omega \ : \ n \mid A > \ .$$

Section 4.3 continues with the formalization of CBabel primitives, describing how coordination contracts are formalized in rewriting logic.

4.3 Coordination Contracts

A coordination contract is a specification of the interaction flow inside a connector and may declare *sequential*, *mutual exclusive* or *guarded* interaction among ports. A *sequential* coordination contract between an input port and an output port specifies that when the the latter is stimulated so is the former. A *mutual exclusion* coordination contract should be declared between two input ports and specifies that only one of these ports is "open" at a time [5]. A *guarded* coordination contract is declared relating an input and an output port. A guarded coordination contract has a *condition*, a *before* block and an *after* block. Once the input port is stimulated and the condition holds, the before block is executed and the output port is stimulated. Once the answer to the output port stimulus arrives, the after block is executed. However, if a message is sent to the input port and the guard condition does *not* hold, that message is *queued* and held until the guard condition turns true.

Before giving the contracts semantics, let us explain the intuition of the formalization. A *sequential* contract between an input port and an output port is a rule rewriting the message representing the port stimulus to the input port to the message representing the output port, also pushing the pair formed by the connector's object identifier together with the output port into the interaction stack, to allow the correct acknowledgment when several output ports are linked to a single input port. The acknowledgment to a synchronous output port, also specified by a rule, pops the top of the interaction and forwards the acknowledgment to the object whose identifier is the first projection of the

[5] Each input port engaged in a mutual exclusion coordination contract has an associated output port, that is, a mutual exclusion of sequential contracts is specified.

new top in the interaction stack. This treatment handles *1:1* or *n:1* interaction styles, that is, when there is a link between one component and one connector or several components and one connector. In the case when a *1:n* interaction style is needed, the sequential contract can be used together with the *parallel* coordination contract, which means that when the input port is unconditionally stimulated the connector's n output ports are also stimulated. This gives rise to a rule rewriting the message representing the stimulus to the input port to n messages representing the stimulus to each of the output ports. If the output ports are synchronous, the treatment for the n messages representing the acknowledgments is to forward one acknowledgment once received all the n acknowledgments. There is no rule for the acknowledgment message if the output port is asynchronous.

A *mutual exclusion* coordination contract, between two synchronous input ports has a binary semaphore semantics, represented by a *status* attribute, that is, a flag attribute, with two rules [6] that are applied *non-deterministically* for the choice of a message to evolve. Once one of the rules is applied, it selects a message from the configuration and the object that represents the connector instance. If the status attribute is *unlocked*, the message is rewritten and the status attribute is set to *lock*, thus preventing the application of one of the choice rules, and therefore the selection of another message to be rewritten. The arrival of an acknowledgment message to the object that represents the connector is also specified by a rules that sets the status attribute to *unlocked*, therefore allowing the rules to be applied non-deterministically again. Mutual exclusion is only allowed between synchronous input ports.

A *guard* contract is formalized by three equations and two rules. One equation is a predicate that evaluates the guard's condition according to the set of attribute values in the object that represents the connector. The other two equations represent the *before* and *after* actions, which are themselves compositions of equations representing the *before* and *after* statements of the guard contract. The first rule specifies that once a message arrives to the input port, *if* the guard condition holds, the *before* equation will be applied to the object that represents the connector and a *send* message is sent to the output port. Otherwise the message to the input port will simply wait *unwritten* in configuration if the guard condition does not hold. This precisely keeps the effect of *holding* a message until the guard is ready to handle it in a way more general than instantiating a queue datatype.

Let us now state these definitions in formal terms. Given a connector declaration (n, V, I, O, c), with n the component's name identifier, V the variable

[6] A different mapping, which would declare the input ports of the connector to be of a subsort of *PortInId* would allow the generation of a single rule.

declaration set, I the input ports declaration set, O the output port declaration set, and c a *sequential contract* declaration, c is a pair of ports (i, o) with the first projection $i \in I$ being an input port and the second projection $o \in O$ being an output port. The declaration of the contract c gives rise to Rule 8 in the rule set R of the associated rewrite theory (Σ, E, R),

(8) $rl\ send(\omega, i, \iota) < \omega\ :\ n \mid A > \Rightarrow\ send(\omega, o, [\omega, o] :: \iota) < \omega\ :\ n \mid A >\ .$

where the operations

$$[\ \_\ ,\ \_\ ] : Oid\ PortOutId \rightarrow OidPortIdPair$$

and

$$\_ :: \_ : Interaction\ Interaction \rightarrow Interaction$$

are constructor operators for sorts *OidPortIdPair* and *Interaction*, respectively, with the sort *OidPortIdPair* being a subsort of *Interaction*, all declared in the rewrite theory *CBABEL-CONFIGURATION*, ω is the object identifier of the object that represents an instance of the CBabel connector, ι is the interaction, and A is the object's attribute set. When the parallel coordination contract is used, Rule 8 has one *send* message for each output port on the right-hand side of the rule. If the output port o is synchronous then Rule 9 is also added to R. When the parallel coordination contract is used, Rule 9 has one *ack* message for each output port on the left-hand side of the rule. Rule 9 is not added if o is asynchronous.

(9) $rl\ ack([\omega, o] :: \iota) < \omega\ :\ n \mid A > \Rightarrow\ ack(\iota) < \omega\ :\ n \mid A >\ .$

Given a connector declaration (n, V, I, O, c), with n the component's name identifier, V the variable declaration set, I the input ports declaration set, O the output port declaration set, and c a *mutual exclusion contract* declaration, c is a four tuple (i_1, o_1, i_2, o_2) with $i_1, i_2 \in I$ and $o_1, o_2 \in O$. The declaration of the contract c gives rise to a class attribute $status\ :\ PortStatus \rightarrow Attribute$ declared in the rewrite theory *CBABEL-CONFIGURATION* and used by each instance ω of class n. Rules 10, 11, 12 and 13 are included in R,

(10) $rl\ send(\omega, i_1, \iota) < \omega\ :\ n \mid status\ :\ unlocked, A > \Rightarrow$
$< \omega\ :\ n \mid status\ : locked, A > send(\omega, o_1, [\omega, o_1] :: \iota)\ .$

(11) $rl\ send(\omega, i_2, \iota) < \omega\ :\ n \mid status\ :\ unlocked, A > \Rightarrow$
$< \omega\ :\ n \mid status\ : locked, A > send(\omega, o_2, [\omega, o_2] :: \iota)\ .$

(12) $rl\ ack([\omega, o_1] :: \iota) < \omega\ :\ n\ |status\ :\ locked, A > \Rightarrow$
$< \omega\ :\ n \mid status\ :\ unlocked, A > ack(\iota)\ .$

(13) $rl\ ack([\omega, o_2] :: \iota) < \omega\ :\ n\ |status\ :\ locked, A > \Rightarrow$
$< \omega\ :\ n \mid status\ :\ unlocked, A > ack(\iota)\ .$

where ω is the object identifier of the object that represents an instance of the

CBabel connector, ι is the interaction, A is the object's attribute set.

Given a connector declaration (n, V, I, O, c), with n the component's name identifier, V the variable declaration set, I the input ports declaration set, O the output port declaration set, and c a *guard contract* declaration, c is a five tuple (i, o, b, β, α) where $i \in I$, $o \in O$, b is a boolean expression, with β and α being sequences of assignment and variable lookup statements or boolean expressions on elements of V. (We shall not give the detailed syntax and meaning of statements and expressions since they are straightforward and here we wish to focus on the meaning of the guard contract. It suffices than to understand β and α as compositions of functions that give meaning to such statements and expressions.) The condition b of c gives rise to a function which is the composition of the statements in b. Moreover, an equation relates the abstract function $opened? : Object \to Bool$ to the function that is the meaning of the guard condition expression b. Functions β and α are represented by the functions $before, after : Object \to Object$, respectively, declared in *CBABEL-CONFIGURATION*. The declarations of β and α give rise to two equations. Each equation is a composition of functions representing the sequence of statements in β and α. (Again, they will not be shown here to keep the presentation focused on the contract's meaning.) Finally, Rules 14 and 15 are added to R.

(14) $$\begin{aligned} &crl\ send(\omega, i, \iota) < \omega : n \mid A > \Rightarrow \\ &before(< \omega : n \mid A >)\ send(\omega, o, [\omega, o] :: \iota) \\ &\ if\ opened?(< \omega : n \mid A >)\ . \end{aligned}$$

(15) $$rl\ ack([\omega, o] :: \iota) < \omega : n \mid A > \Rightarrow\ after(< \omega : n \mid A >)\ ack(\iota)\ .$$

4.4 Application

A CBabel application module declares how the components of an architecture should be put together. It may instantiate components and then link them together by their ports and bind their state variables. (See Section 4.5.)

Formally, a CBabel application module is a triple (x, Y, L, B) where x is the application module's name, Y is the set of instantiation declarations (ω, c) with ω an identifier representing a CBabel component instance and c, also an identifier, representing a CBabel component; L is the set of link declarations $(\omega_1, o, \omega_2, i)$ with ω_1 (resp. ω_2) an identifier representing an instance of c_1 (resp. c_2), o an output port declared in c_1 and i an input port declared in c_2; and B a set of binding declarations which will be formalized in Section 4.5. A CBabel application module gives rise to a rewrite theory (Σ, E, R) such that Σ includes a constant $topology :\to Configuration$ and E includes Equation 16

(16) $$eq\ topology = instantiate\text{-}c_1(\omega_1) \ldots\ instantiate\text{-}c_n(\omega_n)\ .$$

where $n = |Y|$. Each link declaration in L gives rise to an Equation 17 in E.

(17) $$eq\ send(\omega_1, o, \iota) = send(\omega_2, i, \iota)\ .$$

The formalization of bind declarations is given in Section 4.5, next, since they are related to state required variable declarations, subject of that section.

4.5 State Required Variables

State required variables allows for a shared memory communication between a CBabel connector and a CBabel component, that is, if a local variable (declared in a module or connector) changes the state required variable (declared in another connector) bound to the local variable should immediately notice this change, and vice-versa. A bind declaration should be done in the application module relating a variable in a component with a state required variable in a connector.

State required variables are mapped to pairs composed by a value and a status information which could be *changed* or *unchanged*. Bind declarations in the application module are mapped to equations that specify the synchronization between the bound variables. Recall from Section 3 that equations are applied before the rules, therefore the state variables will be synchronized before the rules for messages are applied.

Let us formalize this. Given a CBabel component declaration (n, V, I, O), a state required variable declaration is a pair $required(v, t) \in V$. The declaration of a state required variable in a CBabel component gives rise to an attribute declaration in the class declaration $class\ n \mid v : SateRequired$ in the signature Σ of the associate rewrite theory (Σ, E, R), where *SateRequired* is a sort declared in the rewrite theory *CBABEL-CONFIGURATION* included in Σ. *CBABEL-CONFIGURATION* also declares the constructors $st : T\ Status \rightarrow StateRequired$, for each primitive type T of CBabel, with *Status* having the constructors $changed, unchanged : \rightarrow Status$.

Given a bind declaration $(\omega_1, v_1, \omega_2, v_2, t)$ in a CBabel application module, ω_1 and ω_2 are identifiers representing CBabel components n_1 and n_2, respectively, with $required(v_1, t) \in V_{n_1}$, $(v_2, t) \in V_{n_2}$. A bind declaration gives rise to Equations 18 and 19 in E

(18) $$eq\ < \omega_1 : n_1 \mid v_1 : st(V_1, changed), S_1 > < \omega_2 : n_2 \mid v_2 : V_2, S_2 > = < \omega_1 : n_1 \mid v_1 : st(V_1, unchanged), S_1 > < \omega_2 : n_2 \mid v_2 : V_1, S_2 > \ .$$

(19) $$ceq\ < \omega_1 : n_1 \mid v_1 : st(V_1, unchanged), S_1 >< \omega_2 : n_2 \mid v_2 : V_2, S_2 > = < \omega_1 : n_1 \mid v_1 : st(V_2, unchanged), S_1 >< \omega_2 : n_2 \mid v_2 : V_2, S_2 >$$

if $V_1 \neq V_2$.

where n_1 and n_2 are the CBabel component identifiers with instances ω_1 and ω_2, respectively, V_1 and V_2 are variables of type t, and S_1 and S_2 are the attribute sets of ω_1 and ω_2 respectively.

5 Executing and Verifying CBabel Architecture Descriptions in Maude

The CBabel tool [7] is a prototype executable environment for CBabel that implements the rewriting logic semantics given in Section 4, and allows the execution and verification of CBabel descriptions in the Maude system. As mentioned before, CBabel tool prototype extends Full Maude [10]. Given CBabel descriptions and the rewriting logic semantics presented in Section 4, CBabel tool produces Maude object-oriented modules for each CBabel component and loads this modules into Full Maude modules database. Note that, in this prototype, execution of simulations, that is, rewrites, and the specification of properties to be verified with model checking, for instance, is done using Maude syntax. It is part of our future work create a command interface for CBabel tool that understands *components* and *ports*, and not *objects* and *messages* as the in the current prototype.

In this section we use the CBabel tool to prove properties about producer-consumer-buffer architectures presented in Section 1. We have analyze three properties in the producer-consumer-buffer application: (i) race condition, (ii) deadlock, and (iii), assuming the buffer limited, overflow and underflow, that is, the producer should not add more items than the buffer may hold and the consumer should not remove an item from an empty buffer. Figure 4 shows the execution of the CBabel tool with the architectures from Figures 1, 2 and 3.

The Full Maude command `show module <module> .` pretty-prints the module `<module>` into the screen. Figure 5 shows the rewrite theory generated from the CBabel connector `MUTEX`, given in Figure 2. The first two rules labeled `MUTEX-mutex-out1` and `MUTEX-mutex-out2` are instances of Rules 12 and 13 in Section 4, and the rules labeled `MUTEX-mutex-in1` and `MUTEX-mutex-in2` are instances of Rules 10 and 11, also in Section 4.

To execute or verify an architecture one should manually provide yet another module since the architecture description does not give any specification regarding the internal behavior of the components, the initial state of the system or the properties that should be verified. Moreover, one could also make

[7] The CBabel tool prototype and additional examples may be downloaded from http://www.ic.uff.br/~cbraga/vas/cbabel-tool/.

```
$ maude ../cbabel-tool.maude s-modules.cbabel pc-default.cbabel pc-mutex.cbabel
  pc-guards-mutex.cbabel
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.1.1 built: Jun 15 2004 12:55:31
             Copyright 1997-2004 SRI International
                   Wed Feb  2 18:46:11 2005

               Cbabel Tool 2.3 (February 3th, 2005)

Introduced module CBABEL-CONFIGURATION

rewrites: 1848 in 10ms cpu (0ms real) (184800 rewrites/second)
Introduced module PRODUCER

rewrites: 1848 in 0ms cpu (10ms real) (~ rewrites/second)
Introduced module CONSUMER

...

rewrites: 6134 in 20ms cpu (20ms real) (306700 rewrites/second)
Introduced module PC-GUARDS-MUTEX

Maude>
```

Fig. 4. Running the CBabel tool

```
Maude> (show module MUTEX .)
omod MUTEX is
  including CBABEL-CONFIGURATION .
  class MUTEX | status : PortStatus .
  op mutex@in1 : -> PortInId  [ctor] .
  op mutex@in2 : -> PortInId  [ctor] .
  op mutex@out1 : -> PortOutId [ctor] .
  op mutex@out2 : -> PortOutId [ctor] .

  eq instantiate(O:Oid,MUTEX) = < O:Oid : MUTEX | status : unlocked > .

  rl < O:Oid : MUTEX | status : locked > ack([O:Oid,mutex@out1]:: IT:Interaction)
     => < O:Oid : MUTEX | status : unlocked > ack(IT:Interaction)
     [label MUTEX-acking-mutex@out1] .
  rl < O:Oid : MUTEX | status : locked > ack([O:Oid,mutex@out2]:: IT:Interaction)
     => < O:Oid : MUTEX | status : unlocked > ack(IT:Interaction)
     [label MUTEX-acking-mutex@out2] .
  rl < O:Oid : MUTEX | status : unlocked > send(O:Oid,mutex@in1,IT:Interaction) =>
     < O:Oid : MUTEX | status : locked >
     send(O:Oid,mutex@out1,[O:Oid,mutex@out1]:: IT:Interaction)
     [label MUTEX-sending-mutex@in1] .
  rl < O:Oid : MUTEX | status : unlocked > send(O:Oid,mutex@in2,IT:Interaction) =>
     < O:Oid : MUTEX | status : locked >
     send(O:Oid,mutex@out2,[O:Oid,mutex@out2]:: IT:Interaction)
     [label MUTEX-sending-mutex@in2] .
endom
```

Fig. 5. Rewriting logic semantics for the `MUTEX` connector

verifications using different process scheduling strategies that are, of course, not described at the architecture level. Since CBabel tool does not provide yet a specific syntax for the specification module, we have manually coded one

in the object-oriented rewrite theory VER-PCB, which is presented in Figure 6.

```
(omod APP is inc PC-DEFAULT . endom)

(omod VER-PCB is
 inc APP .
 inc MODEL-CHECKER .

 subsort Configuration < State .

 var C : Configuration . var O : Oid .
 vars IT IT' : Interaction . vars M N : Int .

 rl [producer-do] : done(O, producer@put, IT) => do(O, producer@put, none) .
 rl [consumer-do] : done(O, consumer@get, IT) => do(O, consumer@get, none) .
 rl [buffer-do-put] :
    do(O, buffer@put, IT)
    < O : BUFFER | buffer@items : N , buffer@maxitems : M > =>
    < O : BUFFER | buffer@items : (if s(N) > 3 then 3 else s(N) fi) ,
                   buffer@maxitems : M >
    done(O, buffer@put, IT)  .
 rl [buffer-do-get] :
    do(O, buffer@get, IT) < O : BUFFER | buffer@items : N > =>
    < O : BUFFER | buffer@items : (if (N - 1) < -1 then -1 else (N - 1) fi) >
    done(O, buffer@get, IT) .

 op raceCond : -> Prop .
 eq send(O, buffer@get, IT) send(O, buffer@put, IT') C |= raceCond = true .

 op initial : -> Configuration .
 eq initial = topology do(cons, consumer@get, none) do(prod, producer@put, none) .
endom)
```

Fig. 6. The verification and execution module for the producer-consumer-buffer architectures

The module VER-PCB first includes the modules MODEL-CHECKER and APP, that includes module PC-DEFAULT. The module APP should be redefined to include the CBabel application module that will be verified. This is a simple "interface" to allow us to reuse the verification module. After the inclusion declaration the sort Configuration is declared to be a subsort of sort State, which means that the "soup" of objects and messages will be the states that compose the model that the model checker will verify. Next, the *observable* internal behavior of the objects, that is, a "minimum" specification of the internal behavior necessary to perform the verification task, is specified as four rules. They define that instances of PRODUCER and CONSUMER must produce and consume *continuously* and that instances of BUFFER must increment or decrement its own items variable whenever it receives the buffer@put or buffer@get messages, respectively. For the buffer rule we use a technique called *abstract interpretation* [31]. We need not to use all integers to represent the buffer items. The values -1, buffer@maxitems $+ 1$, and the range $[0,$ buffer@maxitems$]$ suffice. (Actually the range itself could be represented as a constant.) Therefore we only allow the buffer items to be increased up to buffer@maxitems plus one and to be decreased down to -1. Next the raceCond proposition is declared,

representing the race condition property, and is defined as an equation that specifies it as a configuration containing messages `buffer@put` and `buffer@get` simultaneously in the "soup". The initial state of the system was declared by the constant operator `initial`, declared and specified in the application module `PC-DEFAULT`, defined by an equation as the constant operator `topology` plus an initial request to the `PRODUCER` instance `prod` and to the `CONSUMER` instance `cons`.

After entering the `VER-PCB` in the Maude system one may run the model checker to check a formula in linear temporal logic [12] (LTL). Thus, if one reduces the formula `[]~raceCond`, which means that is always true that a race condition will not happen, a counter-example is produced, that is, a path that contains a race condition state is shown. This is reproduced in Figure 7. (The Maude output has been edited since the counter-example is 14 Kbytes long.)

```
Maude> (reduce in VER-PCB : modelCheck(initial,[]~ raceCond) .)
result [ModelCheckResult] :
 counterexample(
  {< buff : BUFFER | buffer@items : 0,buffer@maxitems : 2 >
   < cons : CONSUMER | consumer@get-status : unlocked >
   < default1 : DEFAULT | status : unlocked >
   < default2 : DEFAULT | status : unlocked >
   < prod : PRODUCER | producer@put-status : unlocked >
   do(cons,consumer@get,none)do(prod,producer@put,none),
   'CONSUMER-sending-consumer@get}
  ...
  {< buff : BUFFER | buffer@items : -1,buffer@maxitems : 2 >
   < cons : CONSUMER | consumer@get-status : locked >
   < default1 : DEFAULT | status : unlocked >
   < default2 : DEFAULT | status : unlocked >
   < prod : PRODUCER | producer@put-status : locked >
   send(buff,buffer@get,[default2,default@out]::[cons,consumer@get])
   send(buff,buffer@put,[default1,default@out]::[prod,producer@put]),
   'BUFFER-recevingAndDo-buffer@get}
  ...)
```

Fig. 7. The model checker counter-example for race condition in the `PC-DEFAULT` application.

The search in Figure 8 show that the architecture `PC-DEFAULT` is deadlock-free. The first search in Figure 9 returns a state where the number of items exceeds `buffer@maxitems`, that is, an overflow state. The second search returns a state where the `buffer@items` variable in the buffer is negative representing the underflow condition.

```
Maude> (search in VER-PCB : initial =>! C:Configuration .)

No solution.
```

Fig. 8. Searching for deadlock states in `PC-DEFAULT` application.

As already mentioned in the Section 1, to solve the race condition problem we use a mutual exclusion contract to coordinate the access from the producer and the consumer to the buffer. This leads to the example presented in

```
Maude> (search in VER-PCB : initial =>* C:Configuration
 < buff : BUFFER | buffer@items : N:Int,buffer@maxitems : M:Int >
 such that N:Int > M:Int .)

Solution 1
C:Configuration <- < cons : CONSUMER | consumer@get-status : unlocked >
  < default1 : DEFAULT | status : unlocked >
  < default2 : DEFAULT | status : unlocked >
  < prod : PRODUCER | producer@put-status : locked >
  do(cons, consumer@get,none)
  done(buff,buffer@put,[default1,default@out]::[prod,producer@put]);
M:Int <- 2 ;
N:Int <- 3
=========================================
Maude> (search in VER-PCB : initial =>* C:Configuration
 < buff : BUFFER | buffer@items : N:Int , AS:AttributeSet > such that N:Int < 0 .)

Solution 1
AS:AttributeSet <- buffer@maxitems : 2 ;
C:Configuration <- < cons : CONSUMER | consumer@get-status : locked >
  < default1 : DEFAULT | status : unlocked >
  < default2 : DEFAULT | status : unlocked >
  < prod : PRODUCER | producer@put-status : unlocked >
  do(prod,producer@put,none)
  done(buff,buffer@get,[default2,default@out]::[cons,consumer@get]);
N:Int <- -1
```

Fig. 9. Searching for overflow and underflow states in PC-DEFAULT application.

Figure 2 implemented in the object-oriented rewrite theory PC-MUTEX already introduced in Maude. To be able to execute the model checker in this new architecture one must first redefine the module VER-PCB changing the module PC-DEFAULT for PC-MUTEX in the APP module. After entering the redefined module VER-PCB in the Maude system, one can execute the model checker again to show that now it is always true that a race condition does not happen, as can be seen in Figure 10. Although solving the race condition, the problems of buffer overflow and underflow still exist in this architecture, as shown in Figure 11.

```
Maude> (reduce in VER-PCB : modelCheck(initial, [] ~ raceCond) .)
result Bool :
  true
```

Fig. 10. Model checking for race condition in PC-MUTEX application.

The architecture PC-MUTEX-GUARDS (Figure 3) solves both problems with a mutual exclusive and two guard contracts. One must now redefine the module VER-PCB changing the APP module to include the object-oriented rewrite theory PC-MUTEX-GUARDS. The searches and model checking in Figure 12 show that this new architecture solves the race condition problem, the deadlock, *and* the buffer overflow and underflow problems.

To further exemplify the use of the CBabel tool, we show how a vending machine can be executed and verified in our tool. The CBabel architecture presented in Figures 13 and 14 specifies a concurrent machine to buy cakes

```
Maude> (search [1] initial =>* C:Configuration
              < buff : BUFFER | buffer@items : N:Int , buffer@maxitems : M:Int >
   such that N:Int > M:Int .)

Solution 1
C:Configuration <- < cons : CONSUMER | consumer@get-status : unlocked >
                   < mutx : MUTEX | status : locked >
                   < prod : PRODUCER | producer@put-status : locked >
                   do(cons,consumer@get,none)
                   done(buff,buffer@put,[mutx,mutex@out1]::[prod,producer@put]);
M:Int <- 2 ;
N:Int <- 3
=========================================
Maude> (search [1] initial =>* C:Configuration
         < buff : BUFFER | buffer@items : N:Int , AS:AttributeSet >
   such that N:Int < 0 .)

Solution 1
AS:AttributeSet <- buffer@maxitems : 2 ;
C:Configuration <- < cons : CONSUMER | consumer@get-status : locked >
                   < mutx : MUTEX | status : locked >
                   < prod : PRODUCER | producer@put-status : unlocked >
                   do(prod,producer@put,none)
                   done(buff,buffer@get,[mutx,mutex@out2]::[cons,consumer@get]);
N:Int <- -1
```

Fig. 11. Searching for overflow and underflow states in PC-MUTEX application.

```
Maude> (search [1] initial =>* C:Configuration
         < buff : BUFFER | buffer@items : N:Int , buffer@maxitems : M:Int >
   such that N:Int > M:Int .)
No solution.
=========================================
Maude> (search [1] initial =>* C:Configuration
         < buff : BUFFER | buffer@items : N:Int , AS:AttributeSet >
   such that N:Int < 0 .)

No solution.
=========================================
Maude> (reduce in VER-PCB : modelCheck(initial, [] ~ raceCond) .)
result Bool :
  true
=========================================
Maude> (search initial =>! C:Configuration .)

No solution.
```

Fig. 12. Searching and model checking in PC-MUTEX-GUARDS application.

and apples with dollars and quarters, inspired by the example presented in [8]. A cake costs a dollar and an apple three quarters. One may insert dollars and quarters but the machine only accepts buying cakes and apples with dollars. When one wants to buys an apple, the machine takes a dollar and returns a quarter. The machine can also group four quarters into a dollar. The modules BUY-APPLE, BUY-CAKE, ADD-DOLLAR and ADD-QUARTER represent the concurrent events that may be executed in the machine. The SOLD-APPLE and SOLD-CAKE connectors must guarantee that apples and cakes are sell when pro-

vided enouth money. The `COUNT-DOLLAR` and `COUNT-QUARTER` keep track of the number dollars and quarters in the machine. In the architecture of this example, we used the parallel interaction contract (see Section 4.3), encapsulated in the `SPLIT` connector. Also observe that to describe the interlocking features we used the "alternative" construction, and the "ground" special port in the guarded contracts.

As already mentioned, to verify the architecture one must provide the module that specifies the internal behavior of the components and the initial state of the system. Figure 15 shows the verification and execution module for the vending machine architecture. The two rules define that the `slot` instance of `SLOT` upon receiving a `put-a` or `put-c` message, increment the number of apples or cakes and transforms the `do` message into a `done` message.

Once the `VER-VM` is defined and imported into the Maude system one could perform verification on the architecture. To verify the number of items that can be sold after entering two dollars and two quarters, one may use the Maude search command. In Figure 16 it is shown the search for all final states of the system.

One may also verify if the machine is properly specified. In Figure 17 we present executions of the search command that: (i) validate that the machine does not sell cake if just 3 quarters are provided; and (ii) validate that the amount of dollars or quarters never becomes negative.

6 Related Work

A broad study of the basic concepts of ADLs, their semantics and expressiveness is presented in [35] and [27]. The advantages of having a formal semantics and mechanisms to perform formal analysis on software architectures described by ADLs are also broadly discussed in the literature, for instance in [35], [26] and [13]. Many ADLs such as Rapide [22], Wright [1] and ACME [15] are related to, or are extensions of, existing formalisms and have their semantics expressed in process algebra. These ADLs usually have a supporting environment to ease the modeling and to help in the analysis procedures. For instance, ACME's AcmeStudio allows the modeling of application in a graphical editor tool and also permits some static and semantic verification on the described architectures.

Current research on software architectures are concerned with aspects regarding how to express and verify non-functional aspects [9,18] using ADLs and how to support the described aspects at run-time. Beyond the topology and component makeup, aspects involving complex coordination of interaction components, real-time and QoS becomes part of the architecture

described using ADLs. By using the concept of contracts, CBabel allows the treatment of coordination aspects in a more flexible manner when compared to other ADLs [9]. CBabel also provides QoS contracts that cater for other non-functional aspects [20], which is beyond the present paper's scope.

Our approach has an interesting property of actually executing the CBabel semantics to do the simulations, that is, rewriting a topology, and the verification, since the transformation from CBabel to RWL is the actual *semantics* of CBabel. Moreover, the Maude object-oriented syntax provides an intuitive interpretation for translated CBabel components, which is of easy understanding for most software designers.

Furthermore, an important issue regarding the choice of RWL as underlying framework lies on the fact that it provides an *orthogonal* handling of static aspects of the system, given by equations, and its concurrent behavior, given by rules. This claim is also made in [13], but they use *two* different frameworks, namely equational logic and process algebra. These two aspects are represented in RWL very naturally by equations and rules.

Additionally, the adoption of Maude allows the verification techniques used by our approach to be extended in many different aspects as new improvements are added to this environment, such as real-time features and other verification tools, as mentioned in Section 1, beyond model checking.

7 Final Remarks

In this paper we have given a rewriting logic semantics for CBabel, a software architecture description language. CBabel components are understood as rewrite theories, or more specifically, as object-oriented modules is our Maude implementation. The rewriting logic semantics Maude implementation gave rise to the CBabel tool prototype, which allows CBabel software architecture descriptions to be executed and verified as rewrite theories in Maude. The use of CBabel tool is exemplifies by means of two examples: (i) three variations of the producer-consumer-buffer example with the verification of the properties of race condition, deadlocking, and buffer overflow and underflow; and (ii) a vending machine architecture, verified its correct design.

An important aspect of our translation, that we believe is worth emphasizing, is its *modularity*. In addition of the fact that modularity is an important pragmatic property, we believe it will be quite relevant in the context of architecture reconfiguration [21], an important concept in software architectures that is part of our future work. Given a CBabel component, it can be *completely* translated to rewriting logic without any information about the other components in a given architecture description. The use of *do* and *done* mes-

sages helps in this matter. They allow the *encapsulation* of the treatment for locking and unlocking ports inside the rewrite theory that represents a module. Otherwise, the equations that give semantics to link declarations would be more complex than simply renaming messages: the information about the communication mode of a given port would be necessary in order to lock or unlock a port.

To the best of our knowledge, our approach innovates by devising an executable environment, which includes verification features, for a software architecture language based on its formal semantics. There is, of course, much work ahead, which includes: (i) a complete definition of a command interface that understands software architecture terms such as components and ports and not classes and objects. Also, the answer from the verification tool should be at that level; (ii) our semantics allows one contract per connector. This choice was made to make the semantics simpler. However architectural descriptions become much simpler if a connector is allowed to specify more than one contract. This will be possible in future versions of the tool; (iii) the current concrete syntax of CBabel in the CBabel tool is very close to the *normal form* used by the transformer in the implementation of the CBabel tool, and differs slightly from [36]. In future versions of the tool more flexible declarations will be allowed; (iv) verify more complex architectural descriptions, such as the cruise control example [23]; (v) apply and develop equational abstraction techniques [31] in the context of software architectures.

Acknowledgement

Rademaker would like to thank FGV/EPGE for the partial support. Braga and Sztajnberg would like to acknowledge partial support from CNPq (process PDPGTI 552192/02-3) and FINEP/RNP/CPqD (GIGA - CARAVELA project). Braga is partially sponsored by CNPq (process 300294/2003-4). Sztajnberg acknowledges partial support from FAPERJ/UERJ (Prociência). The authors would like to thank the anonymous reviewers for their fruitful comments. A special thanks is owed to Peter Csaba Ølveczky for his careful review and constructive insights in a draft of this paper.

References

[1] Allen, R. and D. Garlan, *Beyond definition/use: architectural interconnection*, in: *Proceedings of the workshop on Interface definition languages* (1994), pp. 35–45, http://doi.acm.org/10.1145/185084.185101.

[2] Bach, M. J., "The Design of the UNIX Operation System," Prentice-Hall, Englewood Cliffs, NJ, 1987.

[3] Ben-Ari, M., "Principles of Concurrent and Distributed Programming," Prentice-Hall, Englewood Cliffs, NJ, 1990.

[4] Beugnard, A., J.-M. Jzquel, N. Plouzeau and et al, *Making components contract aware*, IEEE Computer (1999), pp. 38–45.

[5] Braga, C. and A. Sztajnberg, *Towards a rewriting semantics to a software architecture description language*, in: A. Cavalcanti and P. Machado, editors, *Proceedings of WMF 2003, 6th Workshop on Formal Methods, Campina Grande, Brazil*, Electronic Notes in Theoretical Computer Science **95** (2003), pp. 148–168.

[6] Bruni, R. and J. Meseguer, *Generalized rewrite theories*, in: J. C. M. Baeten, J. K. Lenstra, J. Parrow and G. J. Woeginger, editors, *Automata, Languages and Programming. 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, Lecture Notes in Computer Science **2719** (2003), pp. 252–266.

[7] Clavel, M., "Reflection in rewriting logic: metalogical foundations and metaprogramming applications," CSLI Publications, 2000.

[8] Clavel, M., F. Durán, S. Eker, N. Martí-Oliet, P. Lincoln, J. Meseguer and C. Talcott, "Maude 2," SRI International and University of Illinois at Urbana-Champaign, http://maude.cs.uiuc.edu (2003).

[9] Dobrica, L. and E. Niemela, *A survey on software architecture analysis methods*, IEEE Transactions on Software Engineering **28** (2002), pp. 638–653.

[10] Durán, F., "Reflective Module Algebra with Applications to the Maude Language," Ph.D. thesis, University of Málaga (1999).

[11] Durán, F. and J. Meseguer, *An extensible module algebra for Maude*, in: *In 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, Electronic Notes in Theoretical Computer Science **15** (1998).

[12] Eker, S., J. Meseguer and A. Sridharanarayanan, *The Maude LTL model checker*, in: F. Gadducci and U. Montanari, editors, *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, Electronic Notes in Theoretical Computer Science **71** (2002), pp. 230–234.

[13] Felix, M. F., "Análise Formal de Modelos de Software Orientados por Abstrações Arquiteturais," Ph.D. thesis, Departamento de Informática, PUC-RJ, Rio de Janeiro, Brazil (2004), in portuguese.

[14] Frölund, S., *Inheritance of synchronization constraints in concurrent object-oriented programming languages*, in: *Proceedings of the ECOOP'92 - 6th European Conference on Object-oriented Programming*, Lecture Notes in Computer Science **615** (1992), pp. 185–196.

[15] Garlan, D., R. Monroe and D. Wile, "Foundations of Component-Based Systems," Cambridge Univ. Press, 2000 pp. 47–68.

[16] Helm, R., I. M. Holland and D. Gangopadhyay, *Contracts: Specifying behavioral compositions in object-oriented systems*, in: *Proceedings of the 4th European Conference on Object-oriented Programming*, Ottawa, 1990, pp. 169–180.

[17] Lea, D., "Concurrent Programming in Java: Design Principles and Patterns," The Java Series, Addison-Wesley, USA, 2003, second edition edition.

[18] Lewis, B. and P. Feiler, *An overview of the SAE architecture analysis design language (AADL) standard: a basis for model-based architecture-driven embedded systems engineering*, in: *Workshop on Architecture Description Languages*, Toulouse, France, 2004.

[19] Löhr, K.-P., *Concurrency annotations for reusable software*, Communications of the ACM **36** (1993), pp. 81–89.

[20] Loques, O., A. Sztajnberg, R. C. Cerqueira and S. Ansaloni, *A contract-based approach to describe and deploy non-functional adaptations in software architectures*, Journal of the Brazilian Computer Society **10** (2004), pp. 5–18.

[21] Loques, O., A. Sztajnberg, J. Leite and M. Lobosco, *On the integration of meta-level programming and configuration programming*, in: *Reflection and Software Engineering (special edition)*, Lecture Notes in Computer Science **1826** (2000), pp. 191–210.

[22] Luckham, D. C. and et al, *Specification and analysis of system architecture using Rapide*, IEEE Transactions on Software Engineering **21** (1995), pp. 336–355.

[23] Magee, J. and J. Kramer, "Concurrency: state models & Java programs," John Wiley & Sons, Inc., UK, 1999.

[24] Martí-Oliet, N. and J. Meseguer, **9**, Kluwer Academic Publishers, Dordrecht, 2002 http://maude.cs.uiuc.edu/papers.

[25] Matsuoka, S. and A. Yonezawa, *Analysis of inheritance anomaly in object-oriented concurrent programming languages*, in: *Research directions in concurrent object-oriented programming*, MIT Press, 1993 pp. 107–150.

[26] Medvidovic, N., D. S. Rosenblum, D. F. Redmiles and J. E. Robbins, *Modeling software architectures in the Unified Modeling Language*, ACM Transactions on Software Engineering and Methodology **11** (2002), pp. 2–57.

[27] Medvidovic, N. and R. N. Taylor, *A framework for classifying and comparing architecture description languages*, in: *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering* (1997), pp. 60–76, http://doi.acm.org/10.1145/267895.267903.

[28] Meseguer, J., *Conditional rewriting as a unified model of concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[29] Meseguer, J., *A logical theory of concurrent objects and its realization in the maude language*, in: *Research directions in concurrent object-oriented programming*, MIT Press, 1993 pp. 314–390.

[30] Meseguer, J., *Membership algebra as a logical framework for equational specification*, in: *In 12th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'97)*, Lecture Notes in Computer Science **1376** (1998), pp. 18–61.

[31] Meseguer, J., M. Palomino and N. Martí-Oliet, *Equational abstractions*, in: F. Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction*, Lecture Notes in Computer Science **2741** (2003), pp. 2–16.

[32] Meseguer, J. and C. Talcott, *Semantic models for distributed object reflection*, in: *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, Lecture Notes in Computer Science **2374** (2002), pp. 1–36.

[33] Meyer, B., *Applying design by contract*, IEEE Computer **25** (1992), pp. 40–51.

[34] Schmidt, D. C., H. Rohnert, M. Stal and D. Schultz, "Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects," John Wiley & Sons, Inc., 2000.

[35] Shaw, M. and D. Garlan, "Software architecture : perspectives on an emerging discipline," Prentice-Hall Inc., USA, 1996.

[36] Sztajnberg, A. and O. Loques, *Reflection in the R-RIO configuration programming environment*, in: *Proceedings of the Middleware'2000 Workshop on Reflective Middleware*, Palisades, NY, USA, 2000, pp. 26–27.

[37] Sztajnberg, A. and O. Loques, *Customizing component-based architectures by contract*, in: W. Emmerich and A. L. Wolf, editors, *Component Deployment, Second International Working Conference*, Lecture Notes in Computer Science **3083**, Edinburgh, UK, 2004, pp. 18–34, iSBN 3-540-22059-3.

```
module BUY-APPLE {
 out port oneway buy-a ;
}

module BUY-CAKE {
 out port oneway buy-c ;
}

module ADD-DOLLAR {
 out port oneway add-$ ;
}

module ADD-QUARTER {
 out port oneway add-q ;
}

module SLOT {
 var int slot@apples ;
 var int slot@cakes ;
 in port oneway put-a ;
 in port oneway put-c ;
}

connector SPLIT {
  in port oneway give-a-in ;
  out port oneway give-a-out ;
  out port oneway return-q ;
  interaction {
   give-a-in > give-a-out | return-q ;
  }
}

connector COUNT-DOLLAR {
 var int dollars = int(0) ;
 in port oneway inc-$ ;
 interaction {
  inc-$ >
  guard( TRUE ) {
   before {
    dollars = dollars + int(1) ;
   }
  } > ground ;
 }
}

connector SOLD-APPLE {
 var int apples = int(5) ;
 staterequired int sa@dollars ;
 in port oneway ack-a ;
 out port oneway give-a ;
 interaction {
  ack-a >
  guard((apples > int(0)) &&
        (sa@dollars > int(0))) {
   before {
    sa@dollars = sa@dollars - int(1) ;
    apples = apples - int(1) ;
   }
  } > give-a ;
 }
}

application VM {
 instantiate BUY-APPLE as ba ;
 instantiate BUY-CAKE as bc ;
 instantiate ADD-DOLLAR as ad ;
 instantiate ADD-QUARTER as aq ;
 instantiate SOLD-CAKE as sc ;
 instantiate SOLD-APPLE as sa ;
 instantiate COUNT-DOLLAR as cd ;
 instantiate COUNT-QUARTER as cq ;
 instantiate SLOT as slot ;
 instantiate SPLIT as split ;
 link ba.buy-a to sa.ack-a ;
 link bc.buy-c to sc.ack-c ;
 link aq.add-q to cq.inc-q ;
 link ad.add-$ to cd.inc-$ ;
 link cq.change to cd.inc-$ ;
 link sa.give-a to split.give-a-in ;
 link split.give-a-out to slot.put-a ;
 link split.return-q to cq.inc-q ;
 link sc.give-c to slot.put-c ;
 bind int sa.sa@dollars to cd.dollars ;
 bind int sc.sc@dollars to cd.dollars ;
}

connector COUNT-QUARTER {
 var int quarters = int(0) ;
 in port oneway inc-q ;
 out port oneway change ;
 interaction {
  inc-q >
  guard( quarters == int(3) ) {
   alternative ( ground ) ;
   before {
    quarters = quarters + int(1) ;
    if ( quarters == int(4) ) {
     quarters = quarters - int(4) ;
    }
   }
  } > change ;
 }
}

connector SOLD-CAKE {
 var int cakes  = int(5) ;
 staterequired int sc@dollars ;
 in port oneway ack-c ;
 out port oneway give-c ;
 interaction {
  ack-c >
  guard((cakes > int(0)) &&
        (sc@dollars > int(0))) {
   before {
    sc@dollars = sc@dollars - int(1) ;
    cakes = cakes - int(1) ;
   }
  } > give-c ;
 }
}
```

Fig. 13. Vending Machine architecture

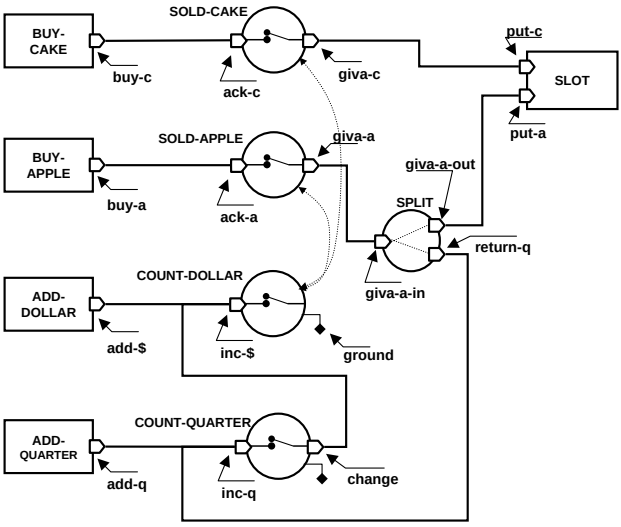

Fig. 14. Vending Machine architecture graphic representation

```
(omod VER-VM is
 inc VM .

 var C : Configuration . var O : Oid .
 var IT IT1 IT2 : Interaction . vars N : Int .
 var P : PortId . var MSG : Msg .

 op initial : -> Configuration .
 eq initial = topology init(do(ad, add-$, none), 2) init(do(aq, add-q, none), 2)
                       init(do(bc, buy-c, none), 2) init(do(ba, buy-a, none), 2) .

 op init : MSG Nat -> Configuration .
 eq init(MSG, N) = MSG init(MSG, N - 1) .
 eq init(MSG, 0) = none .

 rl [slot-put-apple] :
    do(O, put-a, IT) < O : SLOT | slot@apples : N > =>
    done(O, put-a, IT) < O : SLOT | slot@apples : (N + 1) >  .

 rl [slot-put-cake] :
    do(O, put-c, IT) < O : SLOT | slot@cakes : N > =>
    done(O, put-c, IT) < O : SLOT | slot@cakes : (N + 1) >  .
endom)
```

Fig. 15. The verification and execution module for the vending machine

```
Maude> (search initial =>! X:Configuration
  < slot : SLOT | slot@cakes : C:Int , slot@apples : A:Int > .)

Solution 1
A:Int <- 0 ; C:Int <- 2 ;
X:Configuration <- < ad : ADD-DOLLAR | none >
  < split : SPLIT | status : unlocked >
  < aq : ADD-QUARTER | none > < ba : BUY-APPLE | none >
  < bc : BUY-CAKE | none > < cd : COUNT-DOLLAR | dollars : 0,status : unlocked >
  < cq : COUNT-QUARTER | quarters : 2,status : unlocked >
  < sa : SOLD-APPLE | apples : 5,sa@dollars : st(0,unchanged),status : unlocked >
  < sc : SOLD-CAKE | cakes : 3,sc@dollars : st(0,unchanged),status : unlocked >
  send(sa,ack-a,[ba,buy-a]) send(sa,ack-a,[ba,buy-a])

Solution 2
A:Int <- 1 ; C:Int <- 1 ;
X:Configuration <- < ad : ADD-DOLLAR | none > < aq : ADD-QUARTER | none >
  < ba : BUY-APPLE | none > < bc : BUY-CAKE | none >
  < cd : COUNT-DOLLAR | dollars : 0,status : unlocked >
  < cq : COUNT-QUARTER | quarters : 3,status : unlocked >
  < sa : SOLD-APPLE | apples : 4,sa@dollars : st(0,unchanged),status : unlocked >
  < sc : SOLD-CAKE | cakes : 4,sc@dollars : st(0,unchanged),status : unlocked >
  < split : SPLIT | status : unlocked >
  send(sa,ack-a,[ba,buy-a]) send(sc,ack-c,[bc,buy-c])

Solution 3
A:Int <- 2 ; C:Int <- 1 ;
X:Configuration <- < ad : ADD-DOLLAR | none > < aq : ADD-QUARTER | none >
   < ba : BUY-APPLE | none > < bc : BUY-CAKE | none >
   < cd : COUNT-DOLLAR | dollars : 0,status : unlocked >
   < cq : COUNT-QUARTER | quarters : 0,status : unlocked >
   < sa : SOLD-APPLE | apples : 3,sa@dollars : st(0,unchanged),status : unlocked >
   < sc : SOLD-CAKE | cakes : 4,sc@dollars : st(0,unchanged),status : unlocked >
   < split : SPLIT | status : unlocked > send(sc,ack-c,[bc,buy-c])

No more solutions.
```

Fig. 16. Searching for final states in vending machine

```
Maude> (search [1] topology init(do(aq,add-q,none),3) init(do(bc,buy-c,none),1)
  =>* C:Configuration < slot : SLOT | cakes : N:Int , AS:AttributeSet >
  such that N:Int > 1 .)

No solution.
==================================
Maude> (search [1] initial =>* C:Configuration
            < cq : COUNT-QUARTER | quarters : M:Int , AS1:AttributeSet >
            < cd : COUNT-DOLLAR | dollars : N:Int , AS2:AttributeSet >
  such that ((N:Int < 0) or (M:Int < 0)) .)

No solution.
```

Fig. 17. Searching for invalid states of the vending machine