A Rewriting Semantics for a Software Architecture Description Language

Alexandre Rademaker[1] **, Christiano Braga**[1] **, Alexandre Sztajnberg**[2]

[1]Instituto de Computação
Universidade Federal Fluminense

[2]Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro

(arademaker,cbraga)@ic.uff.br, alexszt@ime.uerj.br

Abstract. *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. 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. It is assumed from the reader basic knowledge of algebraic specifications.*

1. Introduction

In [2] the basic ideas of a rewriting logic [16] semantics for the software architecture description language (ADL) CBabel [10] were presented. In this 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 [5]. The CBabel Tool is a direct implementation of the rewriting semantics of CBabel which allows the execution and verification of CBabel descriptions. We shall 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.

Let us begin by recalling the motivation for software architecture description languages and then the syntax of CBabel. The purpose of software architecture description languages is to keep the description of how distributed components are *connected* apart from the descriptions of the internal behavior of each component. Moreover, *connection patterns* may be used to describe how components, that may execute *concurrently*, are linked together. These patterns are called *contracts*. Examples of such contracts are mutual exclusion and sequential coordination. 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 [19], such as components and ports, provides contracts as first class constructions. We will present the syntax of CBabel by means of four variants of the classic producer-consumer-buffer example. Later in Section 4 we will verify these specifications in Maude using the CBabel Tool. It is worth emphasizing that this is *not* the same example used in [2]. It is indeed a *richer* version than that, which uses all features of CBabel. Moreover, the CBabel descriptions are verified using the CBabel Tool, as opposed to [2] where the Maude specifications were used directly. Actually, as shown in Section 4, these Maude specifications are now *automatically* generated by the CBabel Tool.

In the producer-consumer-buffer example there is a producer willing to access a buffer, that 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 than the producer should not add more items than the buffer may hold and the consumer should not remove an item from an empty buffer.

A CBabel architecture that specifies the producer-consumer-buffer configuration is given in Figure 1. Modules specify the component's interfaces that an architecture puts together, which are in this example, `PRODUCER`, `CONSUMER` and `BUFFER`. 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 buffer. 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.[1] A sequence of such port stimuli is called an *interaction*.

Also, 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.

```
module BUFFER {
 var int maxItems = int(2) ;
 var int items = int(0) ;

 in port buffer-put ;
 in port buffer-get ;
}

connector DEFAULT {
 in port def-in ;
 out port def-out ;

 interaction{
   def-in > def-out ;
 }
}

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

```
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.def-in ;
 link default1.def-out to buff.buffer-put ;

 link cons.consumer-get to default2.def-in ;
 link default2.def-out to buff.buffer-get ;
}

module CONSUMER {
 out port consumer-get ;
}
```

Figure 1: Producer-consumer-buffer architecture

As we mentioned before, this architecture has both a race condition problem 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.

```
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 ;
}
```

Figure 2: `MUTEX` connector and the `PC-MUTEX` application

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 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 arrive to the output port, the *after* block of the guarded contract is executed. Another concept that is typically used together with guarded 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 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 means of state variables `items`, `empty` and `full`. The state variables `p-items` and `g-items`, in `GUARD-PUT`

[1] 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.

and GUARD-GET connectors, are bound to items variable in the buffer. Variables g-empty, p-full, and p-maxitems are bound to empty, full, and maxitems variables as described in PC-MUTEX-GUARDS application module. Whenever the number of items exceeds maxItems, the upper bound of the buffer, the state variable full turns true and thus blocks any insertion into the buffer. The opposite happens when the buffer has no items at all and then GUARD-GET blocks any removal from the buffer.

```
connector GUARD-GET {
  var bool full = FALSE ;
  staterequired int g-items ;
  staterequired bool g-empty ;

  in port get-in ;
  out port get-out ;

  interaction {
   get-in >
    guard(full == TRUE) {
     after: {
      g-empty = TRUE ;
      if (g-items == int(0)) {
        full = FALSE ;
      }
     }
    } > get-out ;
  }
}
```

```
connector GUARD-PUT {
  var bool empty = TRUE ;
  staterequired int p-maxitems ;
  staterequired int p-items ;
  staterequired bool p-full ;
  in port put-in ;
  out port put-out ;

  interaction {
   put-in >
    guard(empty == TRUE) {
     after: {
      p-full = TRUE ;
      if (p-items == p-maxitems) {
       empty = FALSE ;
      }
     }
    } > put-out ;
  }
 }
```

```
application PC-MUTEX-GUARDS {
 instantiate BUFFER as buff ;
 instantiate PRODUCER as prod ;
 instantiate CONSUMER as cons ;

 link mutx.mutex-out2 to gget.get-in ;
 link gput.put-out to buff.buffer-put ;
 link gget.get-out to buff.buffer-get ;

 bind int gget.g-items to buff.items ;
 bind int gput.p-items to buff.items ;
 bind bool gput.p-full to gget.full ;
}
```

```
instantiate MUTEX as mutx ;
instantiate GUARD-GET as gget ;
instantiate GUARD-PUT as gput ;

link prod.producer-put to mutx.mutex-in1 ;
link mutx.mutex-out1 to gput.put-in ;
link cons.consumer-get to mutx.mutex-in2 ;

bind bool gget.g-empty to gput.empty ;
bind int gput.p-maxitems to buff.maxItems ;
```

Figure 3: Guarded connectors and the PC-MUTEX-GUARD application

One last example is to allow producers and consumers to be asynchronous. (See Figure 4.) Note that the three architectures shown in Figures 1, 2, and 3 are kept the same. The "unwanted" answers from the GUARD-PUT and GUARD-GET connectors to PRODUCER and CONSUMER, since the output ports of the guards are synchronous, are simply disregarded. Moreover, in this particular example, their properties are also kept in the asynchronous version, that is, the architecture from Figure 1 has both the race condition and the underflow and overflow problems, the architecture in Figure 2 solves the race condition problem, and the architecture in Figure 3 solves both problems.

```
module PRODUCER {
  oneway out port producer-put ;
}
```

```
module CONSUMER {
  oneway out port consumer-get ;
}
```

Figure 4: Asynchronous output port declarations

Rewriting logic is a logic and semantic framework to which several models of computation, logics and specification languages have been mapped to [13] due to its unified view of computation and logic. Maude is one implementation of rewriting logic that realizes it with high-performance. Together with its meta-programming facilities, the Maude system provides a rich tool for the development of formal tools. Moreover, Maude has built-in linear temporal logic model-checking capabilities [7] and several verification tools have been developed for Maude, such as Clavel's Inductive Theorem Prover [4]. Another useful feature of the Maude language is its object-oriented syntax available as object-oriented modules.

The CBabel ADL has a very natural interpretation in object-oriented terms such as components as classes, component's instances as objects, port declarations as messages and port stimulus as message passing [2]. 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 its meta-programming capabilities. This transformation function is the core of the CBabel Tool execution environment prototype.

The rest of the paper is organized as follows. Section 2 gives the necessary background in rewriting logic and object-oriented rewrite theories in Maude. Section 3 gives the rewriting semantics of CBabel. Section 4 presents the execution and verification of CBabel descriptions in Maude using the CBabel Tool. In Section 5 we briefly comment on related works. Section 6 concludes this paper with our final remarks.

2. 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 has the constraint of being *confluent* and *terminating*, which roughly means that every term should have a unique normal form and that should be no infinite chain of rewrites. The rules R should be *coherent*, that is, alternating between equations and rules does not loose 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 [17], a generalization of order-sorted equational logic, is chosen in the Maude system.) Moreover, the notion of *frozen* operators [3] 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.

One of the most useful and important classes 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 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`. It should be noted, however, that *any* algebraic structure can be used to represent a system's concurrent structure. The multiset representation is one particular representation available in the Maude system.

Objects and messages are singleton multiset 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 $\langle\ O : C \mid a_1 : v_1\ ,\ \dots\ ,\ a_n : v_n \rangle$ where O is the object's identifier, C is the object's class identifier, a_i is an object's attribute, and v_i is a_i's corresponding value. The order of the attributes is not relevant, so the `_ , _` operator is also declared with attributes `assoc` and `comm`. Classes are declared in Maude with syntax `class` $C \mid a_1 : s_1\ ,\ \dots\ ,\ a_n : s_n$. 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, with `subclass` syntax (similar to that of `subsort`) 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 would be declared as, `msg to : Oid Oid Data -> Msg`.

The concurrent interactions between objects are axiomatized by rewrite rules. The general form of such a rule is given in Maude as follows:

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

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.

3. An Object-oriented Rewriting Semantics for CBabel Software Architecture Primitives

The fundamental software architecture elements of CBabel could be informally defined as follows: (i) 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; (ii) it is through a *port* that components communicate requesting functionalities or "services" from each other. Ports communicate following a message passing model; (iii) *coordination contracts* define how a group of ports should interact. It may be sequentially, mutually exclusive, or guarded by a condition; (iv) 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; (v) *links* establish the connection of two ports enabling them to interact; (vi) *state required variables* allow for components to exchange information atomically, that is, within a shared-memory model of communication. The following sections formalize these concepts.

3.1. Components

A component can be 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.

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. Moreover, such an object may answer to messages do and $done$. These messages represent, or signalize, the beginning and end of a component's *internal* behavior. Moreover, they carry the sequence of object identifiers in a given *interaction*, that is, a finite sequence of port stimuli from 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 3.2 and the formalization of coordination contracts is given in Section 3.3. 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 of type t which v should be initialized to, and t is the variable's type which must be one of the 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 as a sequence of port stimulus from ports that are related by link declarations, are 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 is 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 Equation 1

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

where ω is an object identifier, 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, $instantiate\text{-}n$ produces an object instance of class n with attributes initialized to the values l declared for the CBabel component variables in V. We continue in Section 3.2 with the formalization of ports.

3.2. Ports

A CBabel component may have input ports or output ports. Input ports are used to provide a service from 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 former 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 3.4 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 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 3.3 will give a detailed explanation on how rules are derived from port and coordination contracts declarations in a connector.

Let us focus then on port declarations in modules. 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*, that is, either a CBabel module or a connector, 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 3.3.

Given a CBabel module declaration (n, V, I, O) or 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 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.

Let us now formalize how port declarations in a CBabel *module* give 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:

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

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

  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:

$$rl\ done(\omega, i, \iota)\ <\omega\ :\ n \mid A>\ \Rightarrow\ <\omega\ :\ n \mid A>\ . \tag{4}$$

- The declaration of a *synchronous output* port $o \in O$ gives rise to Rules 5 and 6,

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

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

  where s is a variable of sort $PortStatus$, declared in the rewrite theory *CBABEL-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:

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

Section 3.3 continues with the formalization of CBabel primitives, describing how coordination contracts are formalized in rewriting logic.

3.3. 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 specifies a "short-circuit" between two ports, that is, given two ports agreeing in a sequential contract, one being an input port and the other an output port, when the input port is unconditionally stimulated, the output port is also stimulated. A *mutual exclusion* coordination contract should be declared between two input ports and specifies that either one or the other port participates in a given interaction. A *guarded* coordination contract is declared relating *synchronous* input and output ports. 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. Messages to the input port of a guarded contract are also queued when the guard is still waiting for an acknowledgment message to its output port.

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 forewords the acknowledgment to the object whose identifier is the first projection of the 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 the first received, ignoring the others. There is no rule for the acknowledgment message if the output port is asynchronous.

A *mutual exclusion* coordination contract, between a synchronous input port and a synchronous output port, has a semaphore semantics and is formalized by two *non-deterministic* rules. Once a rule is applied it selects a message from the configuration to be rewritten and the status attribute is set to $lock$ in the object that represents the connector, thus preventing the application of one of these two rules and therefore the selection of another message to be rewritten. It is also specified by a rule the arrival of an acknowledgment message to the object that represents the connector which sets its status to $unlocked$, therefore allowing the non-deterministic rules to be applied again.

A *guarded* 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 guarded contract. The first rule specifies that once a message arrives to the input port, the *before* equation will be applied to the object that represents the connector setting its status attribute to $lock$ and then a $send$ message is sent to the output port *if* the guard condition holds. Otherwise the message to the input port will simply wait *unwritten* in configuration either if the guard condition does not hold or if an acknowledgment message does not arrive to the output port. 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 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 in the associated rewrite theory (Σ, E, R),

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

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, a conditional version of Rule 9 is added, that forwards only the *first* acknowledgement message and disregards the forthcoming ones. Rule 9 is not added if o is asynchronous.

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

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 and 12 are included in R,

$$\begin{aligned} &rl\ send(\omega, i_1, \iota) < \omega\ :\ n \mid status\ :\ unlocked, A > \Rightarrow \\ &\quad < \omega\ :\ n \mid status\ : locked, A > send(\omega, o_1, [\omega, o_1] :: \iota)\ . \end{aligned} \tag{10}$$

$$\begin{aligned} &rl\ send(\omega, i_2, \iota) < \omega\ :\ n \mid status\ :\ unlocked, A > \Rightarrow \\ &\quad < \omega\ :\ n \mid status\ : locked, A > send(\omega, o_2, [\omega, o_2] :: \iota)\ . \end{aligned} \tag{11}$$

$$\begin{aligned} &rl\ ack([\omega, o] :: \iota) < \omega\ :\ n \mid status\ :\ s, A > \Rightarrow \\ &\quad < \omega\ :\ n \mid status\ :\ unlocked, A > ack(\iota)\ . \end{aligned} \tag{12}$$

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, and s is a variable of sort $PortStatus$.

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 *guarded 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 guarded 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 $opened? : Object \rightarrow Bool$ to the function that is the meaning of the guard condition expression b. Functions β and α are represented by the functions $before, after : Object \rightarrow Object$, 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 13 and 14 are added to R.

$$
\begin{aligned}
&crl\ send(\omega, i, \iota) < \omega : n \mid i\text{-}status : unlocked,\ A > \Rightarrow \\
&\quad before(< \omega : n \mid i\text{-}status : locked,\ A >) \\
&\quad send(\omega, o, [\omega, o] :: \iota) \\
&if\ opened?(< \omega : n \mid i\text{-}status : unlocked,\ A >)\ .
\end{aligned}
\tag{13}
$$

$$
\begin{aligned}
&rl\ ack([\omega, o] :: \iota) < \omega : n \mid i\text{-}status : locked,\ A > \Rightarrow \\
&\quad after(< \omega : n \mid i\text{-}status : unlocked,\ A >)\ ack(\iota)\ .
\end{aligned}
\tag{14}
$$

3.4. 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 3.5.)

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, n) with ω an identifier representing a CBabel component instance and n, also an identifier, representing a CBabel component; L is the set of link declarations $(\omega_1, o, \omega_2, i)$ with ω_i an identifier representing an instance of n_i, o an output port declared in n_1 and i an input port declared in n_2; and B a set of binding declarations which will be formalized in Section 3.5. A CBabel application module gives rise to a rewrite theory (Σ, E, R) such that Σ includes a constant $x :\rightarrow Configuration$ and E includes Equation 15

$$
eq\ x = instantiate\text{-}n_1(\omega_1) \ldots\ instantiate\text{-}n_j(\omega_j)\ . \tag{15}
$$

where $(\omega_i, n_i) \in Y$ and $1 \leq j \leq |Y|$. Each link declaration in L gives rise to a Rule 16 in R.

$$
rl\ send(\omega_1, o, \iota) \Rightarrow send(\omega_2, i, \iota)\ . \tag{16}
$$

The formalization of bind declarations is given in Section 3.5, next, since they are related to state required variable declarations, subject of that section.

3.5. State Required Variables

State required variables allows for a shared memory communication between a CBabel connector and a CBabel component, that is, if a state variable changes in the connector, the other component's variable bound to the connector's state 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 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 2 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 $state(v, t) \in V$. 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, together with constructors $st : T\ Status \rightarrow StateRequired$, for

each primitive type T of CBabel, with $Status$ declared in the rewrite theory *CBABEL-CONFIGURATION* with constructors $changed, unchanged :\rightarrow Status$.

A bind declaration in a CBabel application module is a five tuple $(\omega_1, v_1, \omega_2, v_2, t)$ where 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 17 and 18 in E

$$eq\ < \omega_1\ :\ n_1 \mid v_1\ :\ st(V_1, changed), S_1 >< \omega_2\ :\ n_2 \mid v_2\ :\ V_2, S_2 > = \quad (17)$$
$$< \omega_1\ :\ n_1 \mid v_1\ :\ st(V_1, unchanged), S_1 >< \omega_2\ :\ n_2 \mid v_2\ :\ V_1, S_2 >\ .$$
$$ceq\ < \omega_1\ :\ n_1 \mid v_1\ :\ st(V_1, unchanged), S_1 >< \omega_2\ :\ n_2 \mid v_2\ :\ V_2, S_2 > = \quad (18)$$
$$< \omega_1\ :\ n_1 \mid v_1\ :\ st(V_2, unchanged), S_1 >< \omega_2\ :\ n_2 \mid v_2\ :\ V_2, S_2 >$$
$$if\ V_1 \neq V_2\ .$$

where n_1 and n_2 are the CBabel component identifiers with instances ω_1 and ω_2, respectively, the constructors $st : T\ Status \rightarrow StateRequired$ for $StateRequired$ are declared for each primitive type T of CBabel in the *CBABEL-CONFIGURATION* rewrite theory included in the signature of the rewrite theory associated with the CBabel application module where the bind declarations are given, 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.

4. Executing and Verifying CBabel Descriptions in Maude

The CBabel Tool [2] is a prototype executable environment for CBabel that implements the CBabel's rewriting logic semantics given in Section 3 and allows the execution and verification of CBabel descriptions in the Maude system. The CBabel Tool prototype is a script that calls the Maude system to: (i) given CBabel components descriptions and the rewriting logic semantics presented in Section 3, produces Maude object-oriented modules for each CBabel component, and (ii) load the Maude system with the modules created in (i). Note that in this prototype execution and verification is done at the level of Maude specifications. It is part of our future work to integrate the CBabel Tool in Full-Maude [6], which will allow a complete definition of a proper command interface for the CBabel Tool. One such command example is a verification command that understands *components* and *ports*, and not *objects* and *messages* as the tool currently does.

In this section we will use the CBabel Tool to prove properties about producer-consumer-buffer architectures presented in Section 1. (Not all verifications will be shown due to space limitations but they are available at CBabel Tool web site.) We have at least two problems in the producer-consumer-buffer application: (i) race condition, and (ii) buffer 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 5 shows the execution of the CBabel Tool with the architectures from Figures 1, 2, 3 and 4.

```
$ ./ctp -m cbabel/producer.cbabel -m cbabel/consumer.cbabel
-m cbabel/buffer.cbabel -m cbabel/default.cbabel -m cbabel/mutex.cbabel
-m cbabel/guard-get.cbabel -m cbabel/guard-put.cbabel
-a cbabel/pc-default.cbabel -a cbabel/pc-mutex.cbabel
-a cbabel/pc-mutex-guards.cbabel

CBabel Tool Prototype

    Modules: cbabel/producer.cbabel cbabel/consumer.cbabel
    cbabel/buffer.cbabel cbabel/default.cbabel cbabel/mutex.cbabel
    cbabel/guard-get.cbabel cbabel/guard-put.cbabel

Application: cbabel/pc-default.cbabel cbabel/pc-mutex.cbabel
cbabel/pc-mutex-guards.cbabel

running Maude...

                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
           Maude 2.1.1 built: Jun 15 2004 12:55:31
            Copyright 1997-2004 SRI International
                   Sat Sep  4 18:12:22 2004
Maude>
```

Figure 5: Running the CBabel Tool

The Maude command `show module <module> .` pretty-prints the module `<module>` into the screen. Figure 6 shows the rewrite theory generated from the CBabel connector `MUTEX`, given in Figure 2. The rules labeled `MUTEX-mutex-out1` and `MUTEX-mutex-out2` are instances of Rule 12 in Section 3, and rules labeled `MUTEX-mutex-in1` and `MUTEX-mutex-in2` are instances of Rules 10 and 11 in Section 3, respectively.

[2] The CBabel Tool prototype may be downloaded from `http://www.ic.uff.br/~cbraga/vas/cbabel-tool/`.

```
Maude> show module MUTEX .
mod MUTEX is
  including CBABEL-CONFIGURATION .
  op MUTEX : -> Cid .
  op instantiate-MUTEX : Oid -> Object .
  op mutex-in1 : -> PortInId [ctor] .
  op mutex-in2 : -> PortInId [ctor] .
  op mutex-out1 : -> PortOutId [ctor] .
  op mutex-out2 : -> PortOutId [ctor] .
  eq instantiate-MUTEX (O:Oid) = < O:Oid : MUTEX | none, status : unlocked > .
  rl < O:Oid : MUTEX | status : S:PortStatus, AS:AttributeSet > ack ([O:Oid,
    mutex-out1] :: IT:Interaction) => < O:Oid : MUTEX | status : unlocked,
    AS:AttributeSet > ack (IT:Interaction) [label MUTEX-mutex-out1] .
  rl < O:Oid : MUTEX | status : S:PortStatus, AS:AttributeSet > ack ([O:Oid,
    mutex-out2] :: IT:Interaction) => < O:Oid : MUTEX | status : unlocked,
    AS:AttributeSet > ack (IT:Interaction) [label MUTEX-mutex-out2] .
  rl < O:Oid : MUTEX | status : unlocked, AS:AttributeSet > send (O:Oid,
    mutex-in1, IT:Interaction) => < O:Oid : MUTEX | status : locked,
    AS:AttributeSet > send (O:Oid, mutex-out1, [O:Oid, mutex-out1] ::
    IT:Interaction) [label MUTEX-mutex-in1] .
  rl < O:Oid : MUTEX | status : unlocked, AS:AttributeSet > send (O:Oid,
    mutex-in2, IT:Interaction) => < O:Oid : MUTEX | status : locked,
    AS:AttributeSet > send (O:Oid, mutex-out2, [O:Oid, mutex-out2] ::
    IT:Interaction) [label MUTEX-mutex-in2] .
endm
```

Figure 6: Rewriting logic semantics for the MUTEX connector

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 nor the properties that should be verified. Moreover, one could also make verifications using different process scheduling strategies that are, of course, not described at the architecture level. We have coded such a module in the object-oriented rewrite theory VER-PCB which is presented in Figure 7.

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

omod VER-PCB is
 inc APP . inc MODEL-CHECKER .
 subsort Configuration < State .
 rl done(prod, producer-put, IT) => do(prod, producer-put, none) .
 rl done(cons, consumer-get, IT) => do(cons, consumer-get, none) .
 rl do(O, buffer-put, IT) < O : BUFFER | items : N, MAXITEMS : M, AS >
    =>
    done(O, buffer-put, IT)
    < O : BUFFER | items : (if s(N) > s(M) then s(M) else s(N) fi),
      MAXITEMS : M, AS > [label buffer-do-put] .

 rl do(O, buffer-get, IT) < O : BUFFER | items : N, MAXITEMS : M, AS >
    =>
    done(O, buffer-get, IT)
    < O : BUFFER | items : (if (N - 1) < -1 then -1 else (N - 1) fi),
      MAXITEMS : M, AS > [label buffer-do-get] .
 op raceCond : -> Prop .
 eq send(buff, buffer-put, IT1) send(buff, buffer-get, IT2)
    C:Configuration |= raceCond = true .
 op initial : -> Configuration .
 eq initial = topology
              do(cons, consumer-get, none) do(prod, producer-put, none) .
endom
```

Figure 7: 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 which will be properly defined in the integration of the CBabel Tool with Full-Maude. 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, are specified as four rules. They define that the prod and cons instances of PRODUCER and CONSUMER, respectively, must produce and consume *continuously* and that the BUFFER instance buff 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* [18]. We need not to use all integers to represent the buffer items. The values -1, MAXITEMS $+ 1$, and the range $[0,$ MAXITEMS$]$ suffice. (Actually the range itself could be represented as a constant.) Therefore we only allow the buffer items to be increased up to 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 `VER-PCB` in the Maude system one may run the model checker by executing a reduce command together with a formula in linear temporal logic [7]. Thus, if one reduces the formula `initial |= []~raceCond`, which means that is always true that a race condition will not happen, a counter-example is produced, that is, a path witch contains a race condition state is shown. This is reproduced in Figure 8. (The Maude output has been edited since the counter-example is 14 Kbytes long.) Using the search command

```
reduce in VER-PCB : modelCheck(initial, []~ raceCond) .
rewrites: 76 in 0ms cpu (10ms real) (~ rewrites/second)
result ModelCheckResult: counterexample(...
    {< buff : BUFFER | MAXITEMS : 2,items :
    -1 > < 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,
    def-out] :: [cons,consumer-get]) send(buff, buffer-put, [default1,def-out]
    :: [prod,producer-put]),'BUFFER-send-buffer-get} ...)
```

Figure 8: The model checker counter-example for race condition in the `PC-DEFAULT` application.

one is able to show states where the buffer limits are violated. Figure 9 shows the execution of the search looking for the shortest path to the underflow state.

```
search [1] in VER-PCB : initial =>* C < buff : BUFFER | AS,MAXITEMS : N':Int,
    items : N > such that N < 0 = true .

Solution 1 (state 27)
states: 28  rewrites: 81 in 0ms cpu (0ms real) (~ rewrites/second)
C --> < 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,def-out] :: [cons,consumer-get])
AS --> (none).AttributeSet
N':Int --> 2
N --> -1
```

Figure 9: Searching for an overflow state 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 Figure 2 implemented in the object-oriented rewrite theory `PC-MUTEX` already introduced in Maude System. To be able to execute the model checker in this new architecture one must first redefines 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 may execute the model checker again to show that now is always true that a race condition does not happen (The Maude output is not shown due to its simplicity and the paper space limitations.). Although solving the race condition, the problems of buffer overflow and underflow still exist in this architecture.

The architecture `PC-MUTEX-GUARDS` (Figure 3) solves both problems with a mutual exclusive and 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 10 show that this new architecture solves the race condition problem *and* the buffer overflow and underflow problems.

Finally, to verify the architectures with asynchronous output port declarations in the producer and consumer modules one must run the CBabel Tool with the asynchronous descriptions for these modules as parameters together with applications that include the new versions of the producer and consumer modules. In the asynchronous version one must again manually provide a verification module. New rules for objects behaviors and equation abstractions, specifying that both producer and consumer are now able to produce or consume without waiting for any acknowledgment message.

Due space limitations we will not show here all the verifications for architectures with the asynchronous version of the `PRODUCER` and `CONSUMER` modules. Since in the producer-consumer-buffer example all the properties are kept the same in both asynchronous and synchronous versions, we show in Figure 11 that in the `PC-MUTEX-GUARDS` application with asynchronous versions of `PRODUCER` and `CONSUMER` the race condition and buffer bounded access problems will never occur.

To further exemplify the use of the CBabel Tool, we show how the Tic-Tac-Toe game can be executed and verified in our tool, following the specification in [20] that illustrates how interaction design patterns can be implemented with connectors and contracts.

```
search [1] in VER-PCB : initial =>* C send(buff, buffer-get, IT2) send(buff,
    buffer-put, IT1) .

No solution.
states: 255  rewrites: 4476 in 130ms cpu (130ms real) (34430 rewrites/second)
==========================================
search [1] in VER-PCB : initial =>* C < buff : BUFFER | AS,MAXITEMS : N':Int,
    items : N > such that N > N':Int = true .

No solution.
states: 255  rewrites: 4731 in 130ms cpu (130ms real) (36392 rewrites/second)
==========================================
search [1] in VER-PCB : initial =>* C < buff : BUFFER | AS,MAXITEMS : N':Int,
    items : N > such that N < 0 = true .

No solution.
states: 255  rewrites: 4731 in 130ms cpu (130ms real) (36392 rewrites/second)
==========================================
reduce in VER-PCB : modelCheck(initial, []~ raceCond) .
rewrites: 4481 in 130ms cpu (130ms real) (34469 rewrites/second)
result Bool: true
```

Figure 10: Searching and model checking in `PC-MUTEX-GUARDS` application.

```
search [1] in VER-PCB : initial =>* C send(buff, buffer-get, IT2) send(buff,
    buffer-put, IT1) .

No solution.
states: 255879  rewrites: 17800025 in 520340ms cpu (520430ms real) (34208
    rewrites/second)
==========================================
search [1] in VER-PCB : initial =>* C < buff : BUFFER | AS,MAXITEMS : N':Int,
    items : N > such that N > N':Int = true .

No solution.
states: 255879  rewrites: 18055904 in 678750ms cpu (679030ms real) (26601
    rewrites/second)
==========================================
search [1] in VER-PCB : initial =>* C < buff : BUFFER | AS,MAXITEMS : N':Int,
    items : N > such that N < 0 = true .

No solution.
states: 255879  rewrites: 18055904 in 695160ms cpu (695560ms real) (25973
    rewrites/second)
```

Figure 11: Verification of `PC-MUTEX-GUARDS` application with asynchronous `PRODUCER` and `CONSUMER`.

The CBabel architecture for the Tic-Tac-Toe game is given in Figure 12. The modules `PLAYER`, `GAME` and `DISPLAY` specify the components' interfaces. The `TURN-GUARD1` and `TURN-GUARD2` connectors enforce the alternation of each player in the game. This is accomplished by a simple protocol using two guards and a turn variable, represented by `g1Turn` and `g2Turn` that are bound to each other in the application module. The `OBSERVER`, `SPLITER` and `UPDATER` implement together the *observer* pattern. Additionally, the synchronization between the players, the game and the display is imposed by an interlocking mechanism based on guards. Thus, a player will only be allowed to make his turn after the previous one has been displayed, and in the same way, a display will only occur after a new turn is complete.

As already mentioned, to verify the architecture one should provide the module that specifies the internal behavior of the components and the initial state of the system. Figure 13 shows the object-oriented rewrite theory `VER-TTT` for the *Tic-Tic-Toe* architecture.

The first three rules define that the `p1` and `p2` instances of `PLAYER` must run *continuously* and the `display` instance of `DISPLAY`, upon receiving an `updating` message, simple transforms the `do` message into a `done` message. (The specific internal behavior of the `DISPLAY` module are not relevant). The next rule defines the behavior of the `game` instance of `GAME`, which is similar to the previous ones but upon receiving a `gturn` message it also must store the last player identification on its `lastPlayer` variable. Since CBabel do not know nothing about objects and identifiers, to keep the information about the last player we provide the operation `oidRange` that transforms an object identifier into an integer. This function is used in the verification of the architecture. After that we provide one more equation to consume the acknowledgments from the `upTurnIn` port of `upd` instance of `UPDATER` to the `turnOut2` port of `split` instance of `SPLITER` since the output port `turnOut2` in the `SPLITER` is asynchronous and so does not expect acknowledgment. Finally we declare the initial state of the system.

Once the `VER-TTT` is defined and imported into the Maude system one could perform verifications on the architecture.

Figure 14 presents a search for a state where `game` receives a `gturn` message from the same player that made the last move. Since this search did not find any state, the alternation of players is preserved.

```
module DISPLAY {
 in port updating ;
}

module GAME {
 var int lastPlayer ;
 var int gScore ;
 in port gturn ;
}

module PLAYER {
  out port turn ;
}

connector SPLITER {
  in port turnIn ;
  out port turnOut1 ;
  out port oneway turnOut2 ;

  interaction {
   turnIn > ( turnOut1 | turnOut2 ) ;
  }
}


connector TURN-GUARD1 {
  var int g1Turn = int(1) ;

  in port g1TurnIn ;
  out port g1TurnOut ;

  interaction {
   g1TurnIn >
   guard( g1Turn == int(1) ) {
     after {
      g1Turn = int(2) ;
     }
   } > g1TurnOut ;
  }
}

connector UPDATER {
  staterequired bool upGo ;
  staterequired int upScore ;

  in port upTurnIn ;
  out port update ;

  interaction {
   upTurnIn >
   guard( upGo == TRUE ) {
    after {
     upGo = FALSE ;
    }
   } > update ;
  }
}
```

```
application TIC-TAC-TOE {
 instantiate PLAYER as p1 ;
 instantiate PLAYER as p2 ;
 instantiate TURN-GUARD1 as g1 ;
 instantiate TURN-GUARD2 as g2 ;
 instantiate GAME as game ;
 instantiate DISPLAY as display ;
 instantiate SPLITER as split ;
 instantiate OBSERVER as obs ;
 instantiate UPDATER as upd ;

 link p1.turn to g1.g1TurnIn ;
 link p2.turn to g2.g2TurnIn ;
 link g1.g1TurnOut to obs.obTurnIn ;
 link g2.g2TurnOut to obs.obTurnIn ;
 link obs.obTurnOut to split.turnIn ;
 link split.turnOut1 to game.gturn ;
 link split.turnOut2 to upd.upTurnIn ;
 link upd.update to display.updating ;

 bind int g2.g2Turn to g1.g1Turn ;
 bind bool upd.upGo to obs.obGo ;
 bind int upd.upScore to game.gScore ;
}

connector TURN-GUARD2 {
  staterequired int g2Turn ;

  in port g2TurnIn ;
  out port g2TurnOut ;

  interaction {
   g2TurnIn >
   guard( g2Turn == int(2) ) {
     after {
      g2Turn = int(1) ;
     }
   } > g2TurnOut ;
  }
}

connector OBSERVER {
  var bool obGo = FALSE ;

  in port obTurnIn ;
  out port obTurnOut ;

  interaction {
   obTurnIn >
    guard( obGo == FALSE ) {
     after {
      obGo = TRUE ;
     }
    } > obTurnOut ;
  }
}
```

Figure 12: Tic-Tac-Toe architecture

Two others properties must hold: (i) one turn must be finished before an update message is sent to the display, that is, the updater should only update the display with the status changes; (ii) a new turn must wait until the status of the former one is displayed. The searches in Figure 15 show that these two properties are guaranteed by the architecture.

As a final remark on this section, let us comment on the execution times for our examples. The computer used to perform the verifications was a Pentium III 1 GHz with 1 GB RAM. In the searches applied to the asynchronous producer-consumer-buffer example, for each proposition, Maude searches 255,879 of states in approximately 10 minutes, as opposed to a significantly smaller number of states, and therefore verification times, for the synchronous cases. Also, both the producer-consumer-buffer application and Tic-Tac-Toe game do not leave much room for parallelization and, therefore, a not so large number of rewrite per second since Maude may reach up to millions of rewrites per second [5], in *certain* applications. A more thorough profiling of the rewrite theories generated by the CBabel Tool is ongoing work.

5. Related Work

A broad study of the basic concepts of ADLs, their semantics and expressiveness is presented in [19] and [15]. The advantages of having a formal semantics and mechanisms to perform formal verifications on software architectures described by ADLs are also broadly discussed in the literature, for instance [19], [14] and [8]. Many ADLs such as Rapide [11], Wright [1] and ACME [9] are related to, or are extensions of,

```
omod VER-TTT is
 inc TIC-TAC-TOE .

 rl done(p1, turn, IT) => do(p1, turn, none) .
 rl done(p2, turn, IT) => do(p2, turn, none) .
 rl do(display, updating, IT) => done(display, updating, IT) .

 rl do(game, gturn, IT :: [O, turn])
    < game : Game | lastPlayer : N:Int , AS > =>
    done(game, gturn, IT :: [O, turn])
    < game : Game | lastPlayer : oidRange(O) , AS > .

 op oidRange : Oid -> Int .
 eq oidRange( p1 ) = 1 .
 eq oidRange( p2 ) = 2 .

 eq ack([split, turnOut2] :: IT) = none .

 op initial : -> Configuration .
 eq initial = topology do(p1, turn, none) do(p2, turn, none) .
endom
```

Figure 13: The verification and execution module for the Tic-Tac-Toe game

```
search in VER-TTT : initial =>* C < game : GAME | lastPlayer : N:Int >
                                  send(game, gturn, IT :: [O,turn])
    such that oidRange(O) == N:Int = true .

No solution.
states: 819  rewrites: 13682 in 430ms cpu (500ms real) (31818 rewrites/second)
```

Figure 14: Searching for a violation in the alternation of players

existing formalisms and have their semantics expressed in process algebra. These ADLs usually have a supporting environment to ease the modeling and, some of them, to help in the verification 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.

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.

Also, an important issue regarding the choice of RWL as underlying framework lies on the fact that it provides an *orthogonal* handling of sequential aspects of the system, given by equations, and its concurrent behavior, given by rules. This claim is also made in [8], but they use *two* different frameworks, namely equational logic and process algebra.

Additionally, the adoption of Maude allows the verifications 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.

6. 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, together with a few shell scripts to automate the process of loading modules into the Maude system, gave rise to the CBabel Tool prototype, which allows CBabel software architecture descriptions to be executed and verified as rewrite theories in Maude. We applied the CBabel Tool to four variations of the producer-consumer-buffer example and verified the properties of race condition between the producer and consumer, and buffer overflow and underflow.

An important aspect of our translation, that we believe is worth emphasizing, is its *modularity*. Despite the fact that modularity is an important pragmatic property, we believe it will be quite relevant in the context of architecture reconfiguration [10], 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 $done$ messages 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 rules that give semantics

```
search in VER-TTT : initial =>* C ack([obs,obTurnOut] :: IT1)
                                 send(upd, update, IT2) .

No solution.
states: 819  rewrites: 13682 in 390ms cpu (550ms real) (35082 rewrites/second)
========================================
search in VER-TTT : initial =>* C send(display, updating, IT1)
                                   send(obs, obTurnOut, IT2) .

No solution.
states: 819  rewrites: 13682 in 400ms cpu (540ms real) (34205 rewrites/second)
```

Figure 15: Searching for invalid Display update

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) an integration with Full-Maude to allow 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 allow 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 [21]. In future versions of the tool more flexible declarations will be allowed; (iv) verify more complex architectural descriptions, such as the cruise control example [12]; (v) apply and develop equational abstraction techniques [18] in the context of software architectures.

Acknowledgements 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. Braga is partially sponsored by CNPq process 300294/2003-4. Sztajnberg acknowledges partial support from FAPERJ process APQ1 E26/171430-02 and Prociência.

References

[1] R. J. Allen and D. Garlan. Beyond definition/use: Architectural interconnection. In *IDL Workshop*, number 8 in ACM SIGPLAN Notices, Vol. 29, pages 35–44, August 1994.

[2] C. Braga and A. Sztajnberg. Towards a rewriting semantics to a software architecture description language. In A. Cavalcanti and P. Machado, editors, *Proceedings of WMF 2003, 6o. Workshop de Métodos Formais, Campina Grande, Brazil*, volume 95 of *ENTCS*, pages 148–168. Elsevier, 2003.

[3] R. Bruni and J. Meseguer. Generalized rewrite theories. In *Thirtieth International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

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

[5] M. Clavel, 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.

[6] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Mlaga, Escuela Tcnica Superior de Ingeniera Informtica, 1999.

[7] S. Eker, 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*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[8] M. F. Felix. *Análise Formal de Modelos de Software Orientados por Abstrações Arquiteutrais*. PhD thesis, Informatics Department, PUC-RJ, Rio de Janeiro, Brazil, 2004. in portuguese.

[9] D. Garlan, R. Monroe, and D. Wile. *Foundations of Component-Based Systems*, chapter Acme: Architectural Descriptions of Component-Based Systems, pages 47–68. Cambridge Univ. Press, 2000.

[10] O. Loques, A. Sztajnberg, J. Leite, and M. Lobosco. On the integration of meta-level programming and configuration programming. In *Reflection and Software Engineering (special edition)*, volume 1826 of *Lecture Notes in Computer Science*, pages 191–210, Heidelberg, Germany, June 2000. Springer-Verlag.

[11] D. C. Luckham and et al. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

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

[13] N. Martí-Oliet and J. Meseguer. *Handbook of Philosophical Logic*, volume 61, chapter Rewriting Logic as a Logical and Semantic Framework. Kluwer Academic Publishers, second edition, 2001. `http://maude.cs.uiuc.edu/papers`.

[14] N. Medvidovic, D. Rosdemblum, D. F. Redmiles, and J. E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, 2002.

[15] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the 6th European Software Engineering Conference*, Zurich, Suia, 1997.

[16] J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.

[17] J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, editor, *WADT'97*, volume 1376, pages 18–61. Springer, 1998.

[18] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In F. Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003. Submitted for publication, January 2003.

[19] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall Inc., EUA, 1996.

[20] A. Sztajnberg, M. Lobosco, and O. Loques. Configurando protocolos de intera cão na abordagem R-Rio. In *anais do XIII Simpósio Brasileiro de Engenharia de Software*, pages 29–45, Florianópolis, Brasil, Outubro 1999.

[21] A. Sztajnberg and O. Loques. Reflection in the r-rio environment. In *Proceedings of the Middleware'2000 Workshop on Reflective Middleware*, Palisades, NY, EUA, April 2000.