Towards a Rewriting Semantics for a Software Architecture Description Language

Christiano Braga$^{1,2}$

Universidade Federal Fluminense, Niterói, Brazil

Alexandre Sztajnberg$^{1,3}$

Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brazil

Abstract

Software architecture description languages (ADL) allow a software designer to focus on high-level aspects of an application by abstracting from the details of the components that compose an architecture. It is precisely this abstraction that makes ADLs suitable for verification using model checking techniques. ADLs are, in a way, domain-specific languages for aspects such as coordination, distribution and quality-of-service. The CBabel ADL defines the concept of contracts that precisely captures these architecture-level aspects. In this paper we propose a rewriting semantics for CBabel, that is, a formal semantics for CBabel specified in rewriting logic, a unifying formalism for concurrency models that has interesting properties as a logic and semantic framework due to its unified view of computation and proof. Using the Maude system, a high-performance implementation of rewriting logic, we formally verify the producer-consumer-buffer problem using model checking and state search.

Keywords: Rewriting semantics, architecture description languages, model checking.

1 Introduction

The combination of software architecture and reflection techniques allows the description, deployment and management of software systems in a high level of abstraction, by considering functional components and their interaction

$^1$ Research project sponsored by CNPq under process 552192/2002-3.
$^2$ Researcher sponsored by CNPq under process 300294/2003-4.
$^3$ Researcher sponsored by UERJ\FAPERJ under ProCiencia program.
schemas as separate concerns [10]. Applications designed with this combination in mind usually have a modular and flexible implementation, and can evolve dynamically. This combination also helps the designer to understand the system’s big picture and eases the configuration of the applications to comply with specific non-functional requirements.

Software architectures are described through Architecture Description Languages, often referred by the acronym ADL. Using an ADL the designer can specify the functional composition of a system, selecting modules and associating them to particular interaction styles or contracts. ADLs usually expose module composition, the architecture topology. This explicit exposition facilitates a natural mapping of a described architecture into actual software artifacts and can, in a next stage, facilitate dynamic reconfiguration activities when combined with reflective middleware capabilities [18]. The CBabel ADL [10], in addition to other ADL features, provides the concept of contracts to describe non-functional aspects such as coordination and quality of service (QoS).

Model checking is a formal verification technique used in many application domains [3]. Applying model checking is usually an automatic activity, but the efficiency of the technique depends on the complexity of the application being verified: the technique is computationally expensive and often can lead to a state explosion. However, the abstraction level provided by an ADL description can minimize the state explosion problem. Moreover, the concept of contracts makes CBabel even more appropriate to model checking. The issue then is how to make CBabel descriptions suitable to the application of model checking techniques.

Rewriting logic (RWL) [14] is a unifying formalism for concurrency models that has been proposed as a logic and semantic framework [12], due to its unified view of computation and proof. The Maude system [5] is a high performance implementation of RWL that embeds a model checker.

In this paper we propose a rewriting semantics for CBabel, that is, a formalization of the semantics of CBabel in rewriting logic by means of a systematic mapping form CBabel to RWL. Based on this mapping, a simple, but comprehensive, case study, the producer-consumer-buffer application, is verified in Maude using model checking and state space search.

This paper is organized as follows. In Section 2 we introduce CBabel ADL. In Section 3 we present rewriting logic, and more specifically object-oriented rewrite theories. With these concepts in mind we present, in Section 4, our mapping from CBabel to Maude. The translation of the producer-consumer-buffer is also presented together with its verification using Maude’s model checker and search command. In Section 5 we present some related works.
Finally, in Section 6 we discuss our ongoing work and research perspectives.

2 CBabel

In this Section we present the CBabel (Building Applications by Evolution with Connectors) Architecture Description Language (ADL), a component of the R-RIO reflective middleware. Let us then briefly describe the R-RIO framework before describing CBabel.

The Reflective-Reconfigurable Interconnectable Objects (R-RIO) framework integrates some key concepts of Software Architecture / Configuration Programming (SA/CP) and Meta-Level Programming (M-LP) approaches [10]. This integration helps to achieve separation of concerns, software reuse and support for dynamic configuration.

The first element of the R-RIO framework is a component model based on the following concepts of SA/CP: (i) modules, that are application components that basically encapsulate functional concerns; (ii) connectors, which are used to encapsulate, mediate and handle interaction-domain concerns among modules; (iii) ports, that identify access points through which modules and connectors provide or require services, and are also used to explicitly bind modules and connectors.

The second element of R-RIO is the CBabel ADL used to describe: (i) application’s components, (ii) their interconnection structure or topology, and (iii) contracts specifying non-functional aspects.

The last element of R-RIO is the Configurator: a reflective middleware [18] that provides distributed configuration management and executive services, used to make and control running images from an architecture description.

CBabel, as other ADLs, is a declarative language, which allows the description of software architectures as a composition of selected modules and connectors and their interconnection topology. Additionally it was included in CBabel means to describe non-functional aspects [2] that were identified as recurring in various application domains and should be considered since the design-time: (a) Interaction, to configure the characteristics of module interaction; (b) Distribution, to configure module location and communication; (c) Coordination, to handle module concurrency and synchronization; and (d) QoS, quality of service, regarding operational requirements, such as fault tolerance, security or communication parameters. These non-functional aspects are represented in CBabel as contracts.

To present CBabel’s syntax let us consider a simple version of the producer-consumer-buffer problem. In this version one producer and one consumer try to access a buffer to put and get (respectively) an item. There has to be a
mutual exclusion discipline to which they have to comply with in order to access the buffer in a consistent manner to avoid race conditions. (Actually the example has also an interesting synchronization problem that will not be tackled in this paper.)

First we present the code using a (concurrent) procedural programming language describing the producer using the semaphore abstraction. The ProdConsExample module encapsulates the application’s resources: a semaphore, the producer task, the consumer task and the buffer (the last two ones omitted). The semaphore has to be declared, instantiated and initialized as a shared (global) resource. The producer’s code has to explicitly include the programming of the semaphores to implement the protocols to enter and leave the critical section, where the buffer is accessed in mutual exclusion (function calls `wait(mutex)` and `signal(mutex)` respectively). Clearly code regarding the producer’s functionality is tangled with the code regarding the coordination protocol to access the buffer. (The consumer’s code would be similar.)

```plaintext
Module ProdConsExample;
mutex: semaphore = 1;
task producer: loop
    produces one item; wait (mutex);
    puts item in buffer; signal (mutex);
endloop;
```

Before presenting the same example in CBabel we introduce some of its syntax details. In a module or connector declaration a class name can be defined (and reused) and, optionally, instance references can be assigned in the same declaration. A list of parameters can be specified to be used when the component is loaded. Internally, a module can have one or more ports and internal variables (which will be exposed to the architectural level).

```plaintext
connector | module <class> [((parameter-list))]
            {
             <internal variables> port-list } [instances;]
```

A port in CBabel defines an interface through which a component can provide a service (in port) or require a service (out port). A port type can be declared and reused in other components (if in an outer scope). A port declaration is associated to a port instance reference. This can help automatic context linking of a set of ports. A parameter list and a return type complete the information associated to a port. The semantics of a port is similar to that of methods in object technology: in ports can be mapped to method signatures, and out ports can be mapped to method invocation. The explicit exposure of in and out ports makes the activity of configuring and architecture design more flexible and can facilitate run-time architecture reconfiguration.

```plaintext
in | out port return-type <port-type> (parameter list)[instances];
```
After describing a selection of modules and connectors, either defining classes and/or instances, the initial topology of the architecture can be described. CBabel provides configuration statements to (a) create instances of a module, (b) link out ports of a module to in ports of another module mediated by a connector, (c) start the operation of a module. The syntax of those statements is as follows:

\[
\text{instantiate } \langle \text{mod-class} \rangle \text{ as } \langle \text{instance-name} \rangle \text{ [at } \langle \text{node-name} \rangle \rangle \text{+}
\]

\[
\text{link } \langle \text{client-mod} \rangle[.\langle \text{port} \rangle] \text{ to } \langle \text{server-mod} \rangle[.\langle \text{port} \rangle] \text{ [by } \langle \text{conn} \rangle \rangle \text{+}
\]

\[
\text{start } \langle \text{module} \rangle \text{+}
\]

It should be noted that configuration statements are used to (a) create and constrain module references, (b) define component port bindings and interaction styles and (c) indicate an order to start the operation of a module. The statements are declarative. (At run-time a reflective middleware, part of the R-RIO framework, implements an API with the same interface to actually perform the configuration operations and additionally provides methods to block, unblock and stop architecture components.)

Our initial approach to the producer-consumer-buffer example in CBabel is presented in Figure 1. Three module are declared: BUFFER, PRODUCER and CONSUMER. Each module has a set of ports defined according to its role. For instance, the BUFFER module has two in ports to receive put and get requests from the producer and consumer respectively. In the sequence, the applications topology is described, resulting in module PRODUCER-CONSUMER. Inside this module an instance of each module type is created (\text{instantiate}) and then the modules are interconnected (\text{link}).

```plaintext
module BUFFER {
    in port int (int item) put ;
    in port int (void) get ;
}
module PRODUCER {
    out port int (int item) put
}
module CONSUMER {
    out port int (void) get
}
module PRODUCER-CONSUMER {
    instantiate BUFFER as buff ;
    instantiate PRODUCER as prod ;
    instantiate CONSUMER as cons ;
    link prod.put to buff.put ;
    link cons.get to buff.get ;
}
```

Fig. 1. Producer-consumer-buffer in CBabel

It should be noted that in the PRODUCER-CONSUMER module the producer and consumer modules are linked to the buffer by a default connector, that is, a connector is not explicitly declared in this case. The default connector simply “shortens” the out-in ports.

Up to this point the architecture description defines only a set of components and its interaction topology. The mutual-exclusion is not yet addressed.
To comply with the mutual exclusion coordination requirement a coordination contract will be included in the architecture description. All interaction with the buffer will now be mediated by a single connector that will encapsulate a mutual-exclusion coordination contract.

Figure 2 presents the description of the MUTEX connector type. Concurrent requests arriving to the connector’s in ports (put-in and get-in) are serialized before being forwarded to the respective out ports, put-out and get-out. This message forwarding is declared by means of the interaction contract >. Thus, no concurrent request will get to the buffer. This is accomplished with the exclusive clause defining the out ports that will work in mutual exclusion discipline.

```
connector MUTEX{
  exclusive {
    out port int (int item) put_out ;
    out port int (void) get_out ;
  }
  in port int (int item) put_in ;
  in port int (void) get_in ;
  put_in > put_out ;
  get_in > get_out ;
} mutx
```

```
module PRODUCER-CONSUMER-MUTEX {
  instantiate BUFFER as buff ;
  instantiate PRODUCER as prod ;
  instantiate CONSUMER as cons ;
  link prod.put to buff.put by mutx ;
  link cons.get to buff.get by mutx ;
}
```

Fig. 2. Mutex with CBabel and the new topology

The new topology is defined in the PRODUCER-CONSUMER-MUTEX module also in Figure 2. First, the module instantiation sequence is defined, as in the former topology. Then, the prod instance and the cons instance are linked to the buff instance, mediated by the mutx connector instance (via compatible out-in port bindings). The mutx instance is implicitly created by the Configurator middleware support and does not need to be explicitly instantiated in the description.

### 3 Concurrent Objects in Rewriting Logic

In this section we explain how concurrent object-oriented systems can be modeled in rewriting logic [14]. Meseguer’s seminal publication on the formalization of concurrent objects in rewriting logic can be found in [13] and a more
recent work by Meseguer and Talcott on the modeling of reflective distributed object systems is presented in [15]. The organization of this section is inspired by these two references.

A rewrite theory is a triple $R = (\Sigma, E, R)$, with $(\Sigma, E)$ an equational specification with signature operators $\Sigma$ and a set of equations $E$. The $R$ component is a set of conditional rewrite rules. The static part of a concurrent object system is formalized in rewriting logic as an equational theory $(\Sigma, E)$. The initial model of $(\Sigma, E)$ precisely models the (distributed) states of such a system. The dynamic part of a concurrent object system, that is, the concurrent interactions between objects, is axiomatized in rewriting logic by the rewrite rule set component of a rewrite theory.

The Maude system [5], a high-performance implementation of rewriting logic, provides syntax to represent object-oriented concepts in rewriting logic. Maude’s object-oriented syntax represents the concurrent state, or the system configuration, as a multiset of objects and messages, declared as juxtaposition with the following operator declaration.

\begin{verbatim}
  op _ _ : Configuration Configuration -> Configuration
   [ctor assoc comm id: null] .
\end{verbatim}

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 $(O : C | a_1 : v_1, \ldots, a_n : v_n)$ 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 the _ _, _ operator is also declared with attributes assoc and comm. Classes are declared in Maude with syntax \texttt{class} $C | a_1 : s_1, \ldots, a_n : s_n$ 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, with \texttt{subclass} syntax (similar to that of \texttt{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 \texttt{msg} keyword in a way similar to an operator declaration. For instance, a message named \texttt{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, \texttt{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:

\[
\text{crl [r]} : M_1 \ldots M_n \langle O_{i_1} : F_{i_1} | \text{atts}_{i_1} \rangle \ldots \langle O_{i_k} : F_{i_k} | \text{atts}_{i_k} \rangle \Rightarrow \langle O_{i_1} : F_{i_1}' | \text{atts}_{i_1}' \rangle \ldots \langle O_{i_k} : F_{i_k}' | \text{atts}_{i_k}' \rangle
\]

\[
\langle Q_1 : D_1 | \text{atts}_{1'} \rangle \ldots \langle Q_p : D_p | \text{atts}_{p'} \rangle
\]

\[
M_1' \ldots M_q'
\]

if \( C \).

where \( r \) is the rule label, the \( M \)s are message expressions, \( i_1, \ldots, i_k \), are different numbers among the original \( 1, \ldots, m \), and \( C \) is the rule’s condition.

Rewriting logic then gives a simple inference system to deduce, for a system axiomatized by a rewrite theory \( \mathcal{R} \), all the finitary concurrent computations possible in such a system. Such computations are identified with proofs of the general form \( \alpha : t \rightarrow t' \) in the logic. The intuitive idea is that such proofs/computations correspond to finitary concurrent behaviors of the distributed system so axiomatized. They are described as concurrent rewritings, where several rules may fire simultaneously in the distributed state, can be followed by other simultaneous firing of other rules, and so on.

\section{4 Mapping CBabel to Rewriting Logic}

This section presents a systematic mapping from a significant subset of CBabel to rewriting logic. Our chosen subset represents the most basic concepts in a CBabel description, which are: modules, ports, connectors and the mutual-exclusion contract. We have chosen a synchronous model of communication among ports and the mutual-exclusion coordination contract to represent the concept of contracts in the CBabel subset. More complex contracts, such as quality-of-service, will be studied in the near future.

This section is organized as follows. In Section 4.1 we describe our mapping from CBabel concepts to object-oriented rewriting logic concepts and then in Section 4.2 we apply our mapping to the producer-consumer-buffer example, discussed in Section 2. The formalization of this mapping, as meta-functions in rewriting logic, is ongoing work.

\subsection{4.1 The Mapping}

Our mapping from CBabel to rewriting logic translates CBabel descriptions to object-oriented rewrite theories in rewriting logic following a very simple intuition: CBabel components, that is modules or connectors, are mapped to object-oriented rewrite theories in rewriting logic and CBabel ports are
mapped to messages in rewriting logic. (Table 1 summarizes the mapping.)

\[
\begin{align*}
topology & \rightarrow \text{object-oriented theories} \\
\text{component instance} & \rightarrow \text{object} \\
\text{application state} & \rightarrow \text{multiset of objects} \\
\text{port} & \rightarrow \text{message} \\
\text{link} & \rightarrow \text{unconditional rewrite rule} \\
\text{mutual-exclusion} & \rightarrow \text{non-deterministic conditional} \\
\text{coordination contract} & \rightarrow \text{rewrite rules}
\end{align*}
\]

Table 1
Mapping from CBabel concepts to rewriting logic

As we have seen in Section 2 the basic units of a CBabel description include modules and ports. Roughly speaking, modules are interfaces that wrap around software components providing different levels of functionality such as communication through its ports, declared inside the modules, and coordination of this communication. Ports are the physical channels that make the communication between modules instances. We represent these concepts as object-oriented rewrite theories and messages in rewriting logic, respectively. An instance of a CBabel module is represented as an object in the associated rewrite theory. Messages are parameterized by the object identifiers of the sender and the receiver objects, together with the type of the message. A message type is defined by a module that declares an out port. Message types are used to correctly address the acknowledgment message. (Section 4.2 illustrates this issue when a connector is used to solve the producer-consumer-buffer problem.)

Note that the mapping from modules to object-oriented rewrite theories preserves, in the target rewrite theory, the modularity of the original CBabel description. The modularity property is an important issue in the context of \textit{dynamic reconfiguration} and \textit{extensibility} of architectural descriptions (see Section 2). These concepts will be made precise in rewriting logic when we formalize the R-RIO middleware, part of our future work.

The topology of an application in CBabel is described as components and links between ports attached to components instances. We represent a topology as a multiset of objects in rewriting logic. In CBabel, module instances interact, that is, communicate, synchronously through its linked ports according to the connections defined in the topology. Such a link is mapped to rewriting logic as an unconditional rewrite rule that rewrites the message...
representing the source port to the message representing the target port. We have chosen to represent synchronous communication between ports in rewriting logic by means of acknowledgment messages that are sent back from the target object to the source object that started the communication, in a pure message-passing communication model. Therefore when a module instance starts an interaction, that is, sends a message of a certain type to another module instance, it blocks until the acknowledgment message arrives.

Connectors are also basic units in CBabel descriptions. They are used to mediate the interaction among modules imposing specific interaction policies by means of contracts, such as the mutual-exclusion coordination contract. CBabel connectors, alike CBabel modules, are mapped to object-oriented rewrite theories in rewriting logic. Therefore connectors instances are mapped to objects in rewriting logic. The CBabel mutual-exclusion coordination contract, in the context of a connector, is mapped to non-deterministic conditional rewrite rules in rewriting logic. The mutual-exclusion mapping produces as many rules, in rewriting logic, as connector ports, in CBabel, declared to be involved in the mutual-exclusion contract. The non-deterministic rules choose to rewrite one message among the messages representing the ports under the mutual-exclusion contract. This mapping precisely captures the intuition that the mutual-exclusion contract only allows one port to be accessed at the time.

As a final remark, please note that this mapping can be generalized to the case where several modules are connected to a connector in port, which is a possible description in CBabel. We simply consider this topology as syntactic sugar to the case where the connector has several in ports with each module connected to a single different port.

4.2 The Producer-Consumer-Buffer Example

In Section 2 we described the producer-consumer-buffer topology in CBabel both with and without a connector. In this Section we present a translation of both topologies to Maude. Using Maude’s model checker and search command, we prove that the translation of the architecture description without the connector may lead to a race condition and the translation of the architecture with a connector does not have a race condition.

We begin with the translation of the modules BUFFER, PRODUCER and CONSUMER. Their CBabel descriptions are given in Figure 1. Figure 3 shows the object-oriented rewrite theory produced by the application of the mapping described in Section 4.1 to the CBabel BUFFER description in Figure 1.

Please recall from Section 3 the syntax for object-oriented rewrite theories in Maude. The object-oriented module BUFFER first includes, in protecting mode, that is, preserving the initial model, the module ACK-MSG, responsible
The ack message has three parameters: the object identifier, of sort \textit{Oid}, of the replying object; the object identifier of the object that the \textit{ack} message is being sent to; and the message type. The keyword \texttt{ctor} means that the operator is a constructor. Then the sort \textit{BufferCid} is declared, being a subsort of sort \textit{Cid}. Sort \textit{BufferCid} has only the \texttt{Buffer} constant which is the class identifier for the \texttt{Buffer} class. The sort \textit{BufferOid} is the sort of object identifiers for objects instances of the \texttt{Buffer} class. These declarations are equivalent to the class declarations presented in Section 3. We are not using the simpler class declarations here because they are only available in the Full-Maude language, which is part of the Maude system. Unfortunately for us, Full-Maude did not support all the Maude language constructs, when we started this project, such as rewriting conditions, that is, rewrites in the conditions of conditional rewrite rules, necessary to our mapping.

Since the \texttt{BUFFER} module declares two in ports, namely \texttt{put} and \texttt{get}, two messages, namely \texttt{buffer-put} and \texttt{buffer-get}, are declared to represent these ports. The meaning of having \texttt{buffer-put} or \texttt{buffer-get} in the configuration is that a component is requesting, to the buffer, that a certain data should be put in or retrieved from the buffer, respectively. (Note that the actual data is not relevant to our verification purposes since we want to check architecture-level properties.) After the buffer puts (or gets) the data in (or from) its internal buffer it should return a message acknowledging this action. This is captured by the rules labeled \texttt{buffer-put-ack} and \texttt{buffer-get-ack} in the \texttt{BUFFER} object-oriented Maude module. Modules \texttt{PRODUCER} and \texttt{CONSUMER} have a similar mapping. The difference is that no rewrite rules are generated since they only declare out ports, that is, they only request “services” and do not need to acknowledge to any message.

Please note that the application of the mapping to the modules \texttt{BUFFER},
PRODUCER and CONSUMER is done only once. Their translation is valid both in
the configuration without the connector and in the configuration with the con-
nector. Recall from Section 4.1 that the modularity of the CBabel description
is preserved in rewriting logic.

The CBabel module PRODUCER-CONSUMER describes the topology of the ar-
chitecture of the producer-consumer-buffer example. Figure 4 presents the
object-oriented rewrite theory produced by the application of the mapping to
the CBabel module PRODUCER-CONSUMER. The rewrite theory PRODUCER-CONSUMER
first imports the modules BUFFER, PRODUCER and CONSUMER in protecting mode.
Then it declares three constants, buff, prod, and cons, of sorts BufferOid,
ProducerOid and ConsumerOid, respectively. These constants are object iden-
tifiers for the three objects representing the CBabel module instances in the
producer-consumer-buffer topology. Finally two unconditional rewrite rules
are declared. The first rewrite rule, labeled producer-put-buffer.put, rep-
resents the link between the producer put port and the buffer put port. The
second rule, labeled consumer.get-buffer.get, represents the link between
the consumer get port and the buffer get port. This concludes the application
of the mapping from CBabel to rewriting logic to the CBabel producer-con-
ssumer-buffer topology.

nomod PRODUCER-CONSUMER is
   pr BUFFER . pr PRODUCER . pr CONSUMER .
   op buff : -> BufferOid [ctor] .
   op prod : -> ProducerOid [ctor] .
   op cons : -> ConsumerOid [ctor] .
   rl [producer.put-buffer.put] :
      producer-put(prod, buff, put-msg) =>
      buffer-put(prod, buff, put-msg) .
   rl [consumer.get-buffer.get] :
      consumer-get(cons, buff, get-msg) =>
      buffer-get(cons, buff, get-msg) .
endom

Fig. 4. PRODUCER-CONSUMER object-oriented rewrite theory in Maude

Nevertheless one should manually provide yet another module in order to
execute this specification, since the architectural description does not provide
any specification regarding the internal behavior of the components in the
topology. Since we are interested only on verifying architecture-level properties
we may abstract from the details of the internal behavior of each component.
Moreover, one could also make verifications using different process scheduling
strategies that are, of course, not described at the architecture level but are
platform dependent, that is, rely on a specific operating system or network.
The object-oriented rewrite theory MC-PRODUCER-CONSUMER is presented in
Figure 5.
The module `MC-PRODUCER-CONSUMER` first includes the modules `MODEL-CHECKER` and `PRODUCER-CONSUMER`. After the inclusion declaration the sort `Configuration` is declared to be a subsort of `State`, which means that the object configurations, representing the CBabel topology, will be the states that compose the model that the model checker will verify. Next the `raceCond` proposition is declared, representing the race condition, and is defined by an equation that specifies a race condition as a configuration containing both `buffer-put` and `buffer-get` messages. As we said before, the architecture description does not specify the internal behavior of the modules. Therefore we have added four rewrite rules that specify that the producer and the consumer keep producing and consuming forever. Note, however, that each instance blocks when its action is happening, that is, a producer, for instance, may only produce when the former `producer-put` message in rewriting logic, representing an interaction initiated by the producer through the put port in CBabel, is acknowledged. Finally, the initial configuration is declared as the constant operator `initial`, representing the initial state of the topology with the `init-prod` and `init-cons` messages and the producer, consumer and buffer objects, identified by the object identifiers `prod`, `cons` and `buff`.

After entering these modules in the Maude system we may execute the model checker by executing a reduce command together with a formula in linear temporal logic. Thus if we reduce 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 which contains a race condition...
state is shown. Figure 6 shows this execution in Maude. (The Maude output shown in Figure 6 has been edited to improve readability.)

\[
\text{reduce in MC-PRODUCER-CONSUMER : initial |= []} \overset{\text{raceCond}}{\Rightarrow}.
\]

rewrites: 25 in 0ms cpu (10ms real) (\~{} rewrites/second)

result ModelCheckResult: counterexample({init-prod init-cons

\begin{align*}
&\text{< buff : Buffer | none > < prod : Producer | none >}
&\text{< cons : Consumer | none >,'producer.put} \{\text{init-cons}
&\text{< buff : Buffer | none > < prod : Producer | none >}
&\text{< cons : Consumer | none > producer-put(prod, buff, put-msg),}
&\text{'consumer.get} \} \ldots \{< \text{buff : Buffer | none >}
&\text{< prod : Producer | none > < cons : Consumer | none >}
\end{align*}

buffer-put(prod, buff, put-msg) buffer-get(cons, buff, get-msg), \text{'buffer-put-ack} \ldots \{< \text{buff : Buffer | none >}

\begin{align*}
&\text{< prod : Producer | none > < cons : Consumer | none >}
\end{align*}

ack(buff, prod, put-msg) consumer-get(cons, buff, get-msg), \text{'producer.put.ack})

Fig. 6. Model checking the producer-consumer-buffer topology in Maude

The model checker does not return the shortest counter example path, due to its search strategy. However the shortest path can be obtained using the search command in Maude. Figure 7 presents the execution of the search command looking for the shortest path to the race condition state, which is obtained by the application of the rules labeled \text{producer.put, consumer.get, producer.put-buffer.put, and consumer.get-buffer.get}. (Again we edit the Maude output to improve readability.)

As we have seen in Section 2 one can solve the race condition problem in CBabel by declaring a connector, that we specified in a module named \text{MUTEX}, with the mutual-exclusion coordination contract over its out ports. A new topology should be produced then by linking the producer and the consumer to the mutex in ports and the buffer to the mutex out ports.

Since the modularity of CBabel descriptions is preserved in rewriting logic, we only need to apply the mapping from CBabel to rewriting logic to the \text{MUTEX} CBabel module and to the \text{PRODUCER-CONSUMER-MUTEX} CBabel module.

\[
\text{search [1] in MC-PRODUCER-CONSUMER : initial =>+ C:Configuration}
\]

buffer-put(prod, buff,put-msg) buffer-get(cons, buff, get-msg) .

Solution 1 (state 11)

states: 12 rewrites: 15 in 0ms cpu (0ms real) (\~{} rewrites/second)

\[
\text{< buff : Buffer | none > < prod : Producer | none >}
\]

\[
\text{< cons : Consumer | none > [ label producer.put ]}
\]

[\text{label consumer.get} [ label producer.put-buffer.put

[\text{label consumer.get-buffer.get}

Configuration: \text{< buff : Buffer | none > < prod : Producer | none >}

\[
\text{< cons : Consumer | none > buffer-put(prod, buff, put-msg)}
\]

buffer-get(cons, buff, get-msg)

Fig. 7. Searching for a race condition state in the producer-consumer-buffer
that declares the new topology with the mutex connector. Due to the MUTEX module length, Figure 8 presents only the non-deterministic Maude rewrite rules that formalize, in rewriting logic, the CBabel mutual-exclusion contract for the MUTEX connector.

```plaintext
omod MUTEX is
...
  var M : MutexOid . vars O1 O2 PO PI GO GI : Oid .
  vars T1 T2 : MsgType . var AS : AttributeSet .
  vars C C' : Configuration .
...
  crl [mutex-put-in-mutex-put-out-1] :
    mutex-put-in(PI,M,T1) mutex-get-in(GI,M,T2)
    < M : Mutex | put-in : ( PI , T1 ) , put-out : PO ,
    get-in : ( GI , T2 ) , AS > C =>
    ack(M,PI,T1) mutex-get-in(GI,M,T2)
    < M : Mutex | put-in : ( PI , T1 ) , put-out : PO ,
    get-in : ( GI , T2 ) , AS > C'
    if mutex-put-out(M, PO, T1)
    < M : Mutex | put-in : ( PI , T1 ) , put-out : PO ,
    get-in : ( GI , T2 ) , AS > C =>
    ack(PO, M, T1)
    < M : Mutex | put-in : ( PI , T1 ) , put-out : PO ,
    get-in : ( GI , T2 ) , AS > C' .
...
  crl [mutex-get-in-mutex-get-out-1] :
    mutex-get-in(GI,M,T2) mutex-put-in(PI,M,T1)
    < M : Mutex | get-in : ( GI , T2 ) , get-out : GO ,
    put-in : ( PI , T1 ) , AS > C =>
    ack(M,GI,T2) mutex-put-in(PI,M,T1)
    < M : Mutex | get-in : ( GI , T2 ) , get-out : GO ,
    put-in : ( PI , T1 ) , AS > C'
    if mutex-get-out(M,GO,T2)
    < M : Mutex | get-in : ( GI , T2 ) , get-out : GO ,
    put-in : ( PI , T1 ) , AS > C =>
    ack(GO,M,T2)
    < M : Mutex | get-in : ( GI , T2 ) , get-out : GO ,
    put-in : ( PI , T1 ) , AS > C' .
endom
```

Fig. 8. Excerpt of the MUTEX object oriented rewrite theory in Maude

As opposed to the object representations for the producer, consumer and buffer, the mutex object has four attributes representing the object identifiers of each object connected to its ports. This is necessary in order represent the internal connections inside the connector. The CBabel MUTEX connector has two in ports, namely put-in and get-in and two out ports, namely put-out and get-out. The in ports are in “short circuit” with the out ports. Thus, if the connector did not specify any contract, that is, if it was the so called default connector, the internal connection would be represented simply as two
unconditional rewrite rules that would, each, rewrite a message representing an in port, either put-in or get-in to a message representing an out port, either put-out or get-out.

However, the MUTEX connector specifies that the put-out and get-out ports should be mutually-exclusive. We represent the mutual-exclusion contract by means of non-deterministic conditional rewrite rules. If the in ports are being accessed simultaneously, that is, if there are both a mutex-get-in message and a mutex-put-in message in the configuration, a mutex-put-out or a mutex-get-out message is chosen to be rewritten while the other is put “on hold”. The process of selecting one message “to go through”, that is, to select an out port message to be rewritten, while putting the other “on hold”, is naturally captured by non-deterministic conditional rules together with a rewriting condition in the conditional rule. Note that there should be as many non-deterministic rewrite rules as there are out ports under the mutual-exclusion contract. In the case of the MUTEX module there are two such rules.

The non-concurrent case, not shown in Figure 8, is also specified by conditional rewrite rules. The MUTEX object-oriented rewrite theory has two such rules. The rule for the message mutex-put-in, for instance, rewrites the message mutex-put-out if the message mutex-get-in is not in the configuration, that is, there is no concurrent access to the mutex in ports.

As we said before, the CBabel modules BUFFER, PRODUCER and CONSUMER do not need to be translated again. However, we need to apply the mapping to the CBabel module PRODUCER-CONSUMER-MUTEX that describes the new topology that links the producer and the consumer to the mutex in ports and the mutex out ports to the buffer. In the rewriting logic translation of PRODUCER-CONSUMER-MUTEX we have four rewrite rules rewriting message producer-put to message mutex-put-in, message consumer-get to message mutex-get-in, message mutex-put-out to message buffer-put-in and message mutex-get-out to message buffer-put-out. This module is very similar to the PRODUCER-CONSUMER module in Figure 4 and therefore is not shown here.

As we did for the producer-consumer-buffer topology we need to define a module to specify some (abstract) behavior for the modules. For the new topology producer-consumer-buffer-mutex we do essentially the same as what we did in Figure 5. The differences are the obvious ones: we need to import the MUTEX module and to add a mutex connector object to the initial configuration with the proper object identifiers in its attributes, that is, the prod object identifier assigned to the put-in attribute, cons assigned to the get-in attribute, and buff assigned to both put-out and get-out attributes.
Now we present the execution of the Maude’s model checker and search command with the rewrite theories for the producer-consumer-buffer-mutex topology. First we give the execution of the model checker for the formula initial |= [] ~ raceCond that specifies that is always true that a race condition does not happen. The execution shows that this property holds.

reduce in MC-PRODUCER-CONSUMER-MUTEX : initial |= [] ~ raceCond .
rewrites: 540 in 20ms cpu (20ms real) (27000 rewrites/second)
result Bool: true

Bellow we present a search for a race condition state where both ports of the buffer are being accessed at the same time, that is, both buffer-put and buffer-get messages are in the configuration. The execution shows that there is no such state in the translation of the producer-consumer-buffer-mutex topology.

search [1] in MC-PRODUCER-CONSUMER-MUTEX :
initial =>+ C:Configuration
buffer-put(mutx, buff, put-msg) buffer-get(mutx, buff, get-msg) .
No solution. states: 16 rewrites: 533 in 20ms cpu (20ms real)
(26650 rewrites/second)

Finally, we present the execution of a Maude’ search command showing that it is possible to have both in ports of the mutex being accessed simultaneously, that is, both messages mutex-put-in and mutex-get-in are in the configuration. (The Maude output is, once again, edited to improve readability.)

search [1] in MC-PRODUCER-CONSUMER-MUTEX :
initial =>+ C:Configuration
mutex-put-in(prod,mutx,put-msg) mutex-get-in(cons,mutx,get-msg) .
Solution 1 (state 13)
states: 14 rewrites: 207 in 0ms cpu (10ms real)
< buff : Buffer | none > < prod : Producer | none >
< cons : Consumer | none >
< mutex : Mutex | put-in : (prod,put-msg),
   put-out : buff,
   get-in : (cons, get-msg),
   get-out : buff >
[label producer.put] [label consumer.get] [label prod-mutex-in]
[label cons-mutex-in]
state 13, Configuration:
< buff : Buffer | none > < prod : Producer | none >
< cons : Consumer | none >
< mutex : Mutex | put-in : (prod,put-msg),
   put-out : buff,
   get-in : (cons, get-msg),
   get-out : buff >
mutex-put-in(prod,mutx,put-msg) mutex-get-in(cons,mutx,get-msg)

5 Related Work

Software architecture concepts can be associated to formalisms in many ways [17]. Formal theories can be defined [6], clarifying these concepts or providing
rules to determine if a given architecture is well-formed [16,8]. Some ADLs are proposed as extensions of well known formalisms. Wright ADL [1] is designed as an extension of CSP to specify component interaction using connectors and architectural styles. Rapide ADL [11] emphasizes the behavior of software architectures and simulation to produce refinements.

The advantages of applying the model checking technique to Software Architectures and ADLs has been investigated since these concepts became more mature. For instance, [7,9] present approaches to model check application described with ADLs. Most of these approaches use an ADL derived from a formal specification abstraction such as CSP, Z or Labeled Transition Systems. In our proposal, the CBabel ADL, designed to facilitate the designer’s job to define a software architecture, is systematically transformed to a formal theory specification, which can be model checked. This facilitates the integration of a formal specification in the traditional life-cycle of an application development.

An interesting combination of rewriting logic and Software Architectures is presented in [4, Appendix E]. In this work the authors specify in rewriting logic the semantics for several typical architectural patterns. Meseguer and Talcott present in [15] semantic models of distributed object reflection in rewriting logic, which is related to the R-RIO framework included in our work. Our approach is complementary to these researches for it provides a formal model in rewriting logic of a specific ADL.

6 Conclusion

In this paper we have presented a systematic mapping from a significant subset of the CBabel ADL to object-oriented rewrite theories in rewriting logic. Interesting aspects of our approach are: (i) Architectural descriptions per se are already interesting languages for model checking since they are not wide spectrum languages therefore minimizing the state explosion problem intrinsic to model checking techniques. Moreover, the concept of a contract in CBabel seems to be a good abstraction technique, which is essential to model checking, since it defines what are sometimes called “small languages” or domain-specific languages for different architectural concepts such as coordination or QoS; (ii) Our mapping preserves the modularity of the original descriptions, which is an important issue regarding reconfiguration and extension of architectural descriptions; (iii) Our approach is not an embedding or an extension of a formalism with architectural concepts but is based on a language designed to be an ADL.

Our on going work is focused on the completion of the rewriting seman-
tics to CBabel by mapping the remaining contracts, such as QoS, together with a more complex case study. Next we plan to formalize our mapping as meta-functions in rewriting logic. The implementation of these functions in Maude produces an executable environment for CBabel specifications that should ease the verification process of CBabel specifications. The formalization of the mapping as meta-functions should occur in parallel to the design of the mapping of the Configurator component (see Section 2) of the R-RIO framework.

Acknowledgement

The authors would like to thank Alexandre Rademaker and the anonymous referees for their constructive comments.

References


