A formal tool for specification and analysis of software architectures

Alexandre Rademaker

May 30, 2005

Why study architecture description languages (ADLs)?

Distributed and concurrent applications invariably have coordination requirements (usually attended in the programming phase)

The purpose of ADLs is to keep the description of how distributed components are connected apart from the descriptions of the internal behavior of each component (separation-of-concerns). An ADL has several interesting features such as:

- focus on architecture-level issues.
- modularity of architectural descriptions for easier understanding and better design.
- reuse of components in different architectures.
- may allow dynamic reconfiguration of architectures.

Why CBabel?

CBabel is an ADL that:

- besides components and ports provides contracts as first class constructions for description of coordination requirements.
- described coordination aspects are encapsulated in connectors that mediate all interaction among functional modules.
- so that allows the separation of coordination aspects concerns from functional aspects.

Why a formal semantics for CBabel?

A formal semantics for some architecture description language $\mathcal{L}$ provides:

- an unambiguous definition of what $\mathcal{L}$ means.
- the ability to formally reason about $\mathcal{L}$ and prove desired properties about $\mathcal{L}$-architectures.
- if the specification is executable, the formal reasoning can be computer aided ("interpreter for free").

Formalisms can be used to provide precise, abstract models and to provide analytical techniques based on these models. [Shaw and Garlan]

Rewriting Logic (RWL)

- a logical framework which can represent in a natural way many different logics, languages, operational formalisms, and models of computation.
- its provides an orthogonal handling of static aspects of the system (equations) and its concurrent behavior (rules).
- RWL has a very natural (algebraic) object-oriented representation for distributed systems.
- reflection gives the ability of map $\mathcal{L}$ to RWL as a meta-function for $D_{\mathcal{L}}$ (data-type represent and $\mathcal{L}$-specification) to $\mathcal{R}$ (RWL theory).
- under a quite few assumptions RWL is executable. Specifications in RWL are executable with CafeOBJ, ELAN, and Maude.

Why Maude?

- is one implementation of RWL with high-performance and with meta-programming facilities;
- built-in verification tools: breath-first search, and LTL model-checker. Several others available through reflection: Inductive Theorem Prover, Church-Rosser Checker, Sufficient Completeness Checker, and Real-Time Maude; (inter-operation of verification tools)
- has an object-oriented syntax available as object-oriented modules;
- is well-suited to specify concurrent and distributed object-based systems;

An example of OO Maude specification I

```
omod BANK-ACCOUNT is
  including CONFIGURATION .
  protecting INT .
  class Account | bal : Int .
  msgs credit debit : Oid Int -> Msg .

  var O : Oid . vars N M : Int .
  rl [credit] :
    < O : Account | bal : N > credit(O, M)
    =>
    < O : Account | bal : N + M > .
  crl [debit] :
    < O : Account | bal : N > debit(O, M)
    =>
    < O : Account | bal : N - M >
  if N >= M .
endom
```

An example of OO Maude specification II

- state space/data types defined by algebraic specifications (equations).
- behavior defined by rewriting rules.
- a computation is a sequence of rewrite steps from an initial. state.

The VAS project

Our work is in the context of the VAS (Verification of Software Architectures) project. Objectives:

- Specify a modular rewriting semantics to CBabel ADL and the R-RIO framework;
- Verify CBabel descriptions using model checking techniques;

Software Architecture elements of CBabel

- A component can be either a module or a connector. A module is a “wrapper” to an entity that performs a computation. A connector mediates the interaction among modules;
- It is through a port that components communicate requesting functionalities or “services” from each other;
- Coordination contracts define how a group of ports should interact;
- Links establish the connection of two ports;
- State required variables allow for components to exchange information atomically (shared-memory model);
- An application is a special module that declares how each component should be instantiated and linked, and how state variables should be bound to each other;

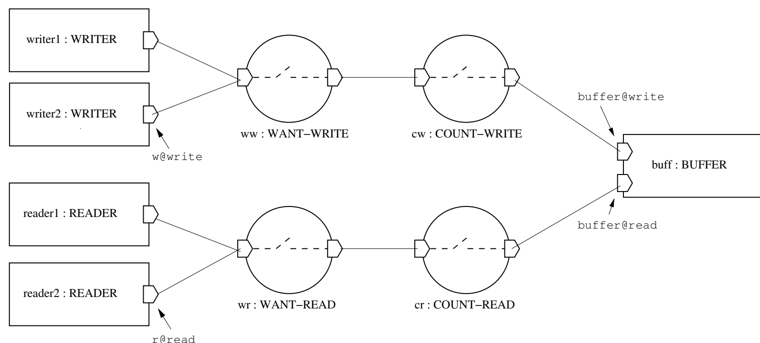
The Readers and Writers example

There is a data area shared among a number of processes, named buffer. There are a number of processes that only read the buffer (readers) and a number that only write to the buffer (writers). The following conditions must be satisfied:

- Any number of readers may simultaneously read the buffer;
- Only one writer at a time may write to the buffer;
- If a writer is writing to the buffer, no reader may read it.

A CBabel solution for the Readers and Writers

The informal graphic representation:



A CBabel solution for the Readers and Writers

```
module READER {
 out port r@read ;
}

module WRITER {
 out port w@write ;
}

module BUFFER {
 in port buffer@read ;
 in port buffer@write ;
}
```

```
connector WANT-WRITE {
 var int want-write = 0 ;
 in  port in-want-write ;
 out port out-want-write ;

 interaction {
   in-want-write >
     guard( TRUE ) {
       before {
         want-write = want-write + 1 ;
     }}
   > out-want-write ;
}}
```

A CBabel solution for the Readers and Writers

```
connector COUNT-WRITE {
 var bool writing = FALSE ;
 staterequired int cw@want-read ;
 [...]
 in  port in-count-write ;
 out port out-count-write ;

 interaction {
  in-count-write > guard(
   (cw@readers == 0 && writing == FALSE) &&
    (cw@want-read == 0 || cw@turn == 1)){
   before {
     cw@want-write = cw@want-write - 1 ;
     writing = TRUE ;
   }
   after {
     writing = FALSE ;
     cw@turn = 0 ;
  }} > out-count-write ;
}}
```

A CBabel solution for the Readers and Writers

```
application READERS-WRITERS {
 instantiate WRITER      as writer1 ;
 instantiate WANT-WRITE  as ww ;
 instantiate COUNT-WRITE as cw ;
 instantiate BUFFER      as buff ;
 [...]

 link writer1.w@write    to ww.in-want-write ;
 [...]

 bind int  cw.cw@want-read  to wr.want-read ;
 [...]
}
```

Mapping CBabel to RWL

The CBabel concepts have a natural interpretation in object-oriented terms:

component	$\longrightarrow$	class
component instance	$\longrightarrow$	object
application state	$\longrightarrow$	multiset of objects/messages
port declaration	$\longrightarrow$	message declaration
port stimulus	$\longrightarrow$	message passing (rewrite rule)
link	$\longrightarrow$	unconditional equation
coordination contract	$\longrightarrow$	rewrite rules
local/state required variable	$\longrightarrow$	class attribute
bind of variables	$\longrightarrow$	equations

Maude CBabel tool

- The Maude CBabel tool is a direct implementation of the rewriting semantics of CBabel which allows the execution and analysis of CBabel descriptions;
- Given a term that represents a CBabel component description and the rewriting logic semantics presented previously, the meta-function `cb2mod` produces a term that represents a Maude OO module;
- It extends the Full Maude environment allowing one to direct import CBabel description into the Full Maude's database of modules and theories;

Mapping CBabel to RWL: READER module

```
omod READER is
  including CBABEL-CONFIGURATION .
  class READER | r@read-status : PortStatus .
  op r@read : -> PortOutId [ctor] .

  eq instantiate(O:Oid,READER) =
     < O:Oid : READER | none,r@read-status : unlocked > .

  rl [READER-sending-r@read] :
   do(O:Oid,r@read,none) < O:Oid : READER | r@read-status : unlocked >
   =>
   send(O:Oid,r@read,[O:Oid,r@read])
   < O:Oid : READER | r@read-status : locked > .

  rl [READER-receivingAck-r@read] :
    ack([O:Oid,r@read]) < O:Oid : READER | r@read-status : locked >
    =>
    done(O:Oid,r@read,none)
    < O:Oid : READER | r@read-status : unlocked > .
endom
```

Mapping CBabel to RWL: COUNT-WRITE connector I

```
omod COUNT-WRITE is
  including CBABEL-CONFIGURATION .
  class COUNT-WRITE | cw@readers : StateRequired,
         cw@turn : StateRequired, cw@want-read : StateRequired,
         cw@want-write : StateRequired, writing : Bool .

  op in-count-write  : -> PortInId  [ctor] .
  op out-count-write : -> PortOutId [ctor] .

  op get-cw@readers : Object -> Int .
  op set-cw@readers : Object Int -> Object .
  eq get-cw@readers(< O:Oid : COUNT-WRITE |
                      cw@readers : st(V:Int,S:Status)>) = V:Int .
  eq set-cw@readers(< O:Oid : COUNT-WRITE |
                      cw@readers : st(V:Int,S:Status)>, V':Int)
    = < O:Oid : COUNT-WRITE | cw@readers : st(V':Int,changed)> .
  [...]
```

Mapping CBabel to RWL: COUNT-WRITE connector II

```
eq instantiate(O:Oid,COUNT-WRITE)
  = < O:Oid : COUNT-WRITE | writing : false ,
      cw@readers : st(0,unchanged) , cw@turn : st(0,unchanged) ,
      cw@want-read : st(0,unchanged) ,
      cw@want-write : st(0,unchanged) > .

ceq after(OBJ:Object)
  = set-cw@turn(set-writing(OBJ:Object,false),0)
 if class(OBJ:Object)= COUNT-WRITE .

ceq before(OBJ:Object)
  = set-writing(set-cw@want-write(OBJ:Object,
                get-cw@want-write(OBJ:Object)- 1),true)
 if class(OBJ:Object)= COUNT-WRITE .

ceq open?(OBJ:Object)
  = (get-cw@readers(OBJ:Object)== 0
     and get-writing(OBJ:Object)== false) and
    (get-cw@want-read(OBJ:Object)== 0
```

Mapping CBabel to RWL: COUNT-WRITE connector III

```
      or get-cw@turn(OBJ:Object)== 1)
  if class(OBJ:Object)= COUNT-WRITE .

 rl [COUNT-WRITE-acking-out-count-write] :
   < O:Oid : COUNT-WRITE | >
   ack([O:Oid,out-count-write]:: IT:Interaction) =>
   after(< O:Oid : COUNT-WRITE | >) ack(IT:Interaction) .

 crl [COUNT-WRITE-sending-in-count-write] :
   < O:Oid : COUNT-WRITE | >
   send(O:Oid,in-count-write,IT:Interaction) =>
   before(< O:Oid : COUNT-WRITE | >)
   send(O:Oid,out-count-write,[O:Oid,out-count-write]:: IT:Interaction)
 if open?(< O:Oid : COUNT-WRITE | >)= true .
endom
```

Mapping CBabel to RWL: application module I

```
omod READERS-WRITERS is
  including CBABEL-CONFIGURATION .
  including WRITER .
  [...]

  ops buff cr cw reader1 reader2 wr writer1 writer2 ww : -> Oid .

  op topology : -> Configuration .
  eq topology = instantiate(buff,BUFFER) instantiate(cr,COUNT-READ)
                instantiate(cw,COUNT-WRITE) [...] .
```

Mapping CBabel to RWL: application module I

```
  eq < cr : COUNT-READ | cr@want-read : st(V1:Int,changed)>
     < wr : WANT-READ | want-read : V2:Int > =
     < cr : COUNT-READ | cr@want-read : st(V1:Int,unchanged)>
     < wr : WANT-READ | want-read : V1:Int > .
  ceq < cr : COUNT-READ | cr@want-read : st(V1:Int,unchanged)>
      < wr : WANT-READ | want-read : V2:Int > =
      < cr : COUNT-READ | cr@want-read : st(V2:Int,unchanged)>
      < wr : WANT-READ | want-read : V2:Int >
  if V1:Int =/= V2:Int = true .
  [...]

  eq send(writer1,w@write,IT:Interaction) =
     send(ww,in-want-write,IT:Interaction)  .
  [...]
endom
```

Characteristics of the mapping

The rewriting semantics that we have given to CBabel:

- uses the object-oriented notation for rewriting logic providing intuitive interpretation easy understanding;
- preserves the modularity of CBabel in the analysis (1-1 relation between objects and CBabel module's instances);
- simplicity is obtained by allowed just one contract peer connector;

The execution module for Readers and Writers

```
(omod RW-EXEC is inc READERS-WRITERS .
 var O : Oid . var IT : Interaction .
 rl [reading] :
    < O : READER | > done(O, r@read, IT)  =>
    < O : READER | > do(O, r@read, none) .
 rl [writing] :
    < O : WRITER | > done(O, w@write, IT) =>
    < O : WRITER | > do(O, w@write, none) .
 rl [buffer-write] :
    < O : BUFFER | > do(O, buffer@write, IT) =>
    < O : BUFFER | > done(O, buffer@write, IT) .
 rl [buffer-read] :
    < O : BUFFER | > do(O, buffer@read, IT)  =>
    < O : BUFFER | > done(O, buffer@read, IT) .
 op initial : -> Configuration .
 eq initial = topology
              do(writer1, w@write, none) do(writer2, w@write, none)
              do(reader1, r@read, none) do(reader2, r@read, none) .
endom)
```

The analysis module for Readers and Writers

```
(omod RW-VER is
 inc RW-EXEC .
 inc MODEL-CHECKER .
 subsort Configuration < State .
 var C  : Configuration .  var    O : Oid . var P : PortOutId .
 var IT : Interaction .    vars N M : Int .

 ops writing reading : Oid -> Prop .
 eq C < buff : BUFFER | > send(buff, buffer@write, IT :: [O, P])
    |= writing(O) = true .
 eq C < buff : BUFFER | > send(buff, buffer@read, IT :: [O, P])
    |= reading(O) = true .

 ops writing reading : -> Formula .
 eq writing = (writing(writer1) \/ writing(writer2)) .
 eq reading = (reading(reader1) \/ reading(reader2)) .
 op raceCond : -> Formula .
 eq raceCond = ((writing(writer1) /\ writing(writer2)) \/
               (writing /\ reading)) .
endom)
```

Examples of analysis in Readers and Writers

Is it always true that the race condition will not happen?

```
Maude> (red modelCheck(initial, [] ~ raceCond) .)
result Bool :
  true
```

Is it possible two readers to be, simultaneously, reading the buffer?

```
Maude> (search [1] initial =>* C:Configuration
                   send(buff, buffer@read, IT1:Interaction)
                   send(buff, buffer@read, IT2:Interaction) .)
Solution 1
C:Configuration <- < buff : BUFFER | none > [...]
IT1:Interaction <-
 [cr,out-count-read]::[wr,out-want-read]::[reader1,r@read];
IT2:Interaction <-
 [cr,out-count-read]::[wr,out-want-read]::[reader2,r@read]
```

Others examples and results

- Others concurrent algorithms were implemented: vending machine, producers and consumers, dining philosophers, more analysis of readers and writers.
- During the implementations:
  - vending machine: compare abstraction level of CBabel vs. Maude specification.
  - producers and consumers: abstraction and modular analysis to avoid state space explosion.
  - readers and writers: fairness and the analysis of starvation with the LTL model checker.
  - philosophers: how unfair rewrites collaborate with the state explosion problem and an over-specification.

Future work

- pretty-print of analysis results only at maude level and also the analysis modules must be defined at Maude level.
- a more robust user interface at CBabel domain could be defined.
- investigate the use of the Palomino's abstract generation (some examples were already done).
- the Verdejo's strategies language could be use to control the fairness.
- integration with others available tools such as Real-Time Maude (to analysis of QoS contracts).

Contributions and final remarks

- the tool is quite useful to different kinds of analysis.
- modularity allows the scalability of the analysis for more complex architectures.
- RWL provides orthogonal handling of static aspects of the system (equations) and its concurrent behavior (rules).
- contributions for the (re)design of CBabel language.