A Rewriting Semantics for a Software Architecture Description Language

Alexandre Rademaker
Universidade Federal Fluminense
arademaker@ic.uff.br

Christiano Braga
Universidade Federal Fluminense
cbraga@ic.uff.br

Alexandre Sztajnberg
Universidade do Estado do Rio de Janeiro
alexszt@ime.uerj.br

Acknowledgements

CNPq and EPGE-FGV for partial support.

Outline

- Motivation
- Rewriting Logic and Maude
- CBabel ADL
- Mapping CBabel to RWL
- The CBabel Tool
- Producer-Consumer-Buffer example
- Turn-based game example
- Developments and future work
- Contributions

Motivation

- The purpose of architecture description Languages (ADLs) is to keep the description of *how* distributed *components* are *connected* apart from the descriptions of the *internal behavior* of each component;
- *Connection patterns* may be used to describe how components, that may execute *concurrently*, are linked together;
- 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 architectures;
  - If the specification is *executable*, the formal reasoning can be *computer aided*;

Rewriting Logic (RWL)

- A logical framework which can represent in a natural way many different logics, languages, operational formalisms, and models of computation;
- Specifications in rewriting logic are *executable* with CafeOBJ, ELAN, and Maude;
- Maude...
  - is one implementation of RWL with high-performance and with meta-programming facilities;
  - has a built-in LTL model-checker and several others verification tools: breadth-first search, theorem prover, and Church-Rosser checker;
  - has an object-oriented syntax available as object-oriented modules.
  - is well-suited to specify concurrent and distributed object-based systems;

Maude

Maude specifications are *rewriting logic* theories:

- State space/data types defined by algebraic specifications (equations);
- Dynamic behavior defined by rewriting rules;
- A behavior is a sequence of rewrite steps from an initial state;
- The deduction rules of the logic defines which sequence hold: $R \vdash [t]_E \rightarrow [t']_E$ if state $[t]_E$ can be rewrite to $[t']_E$ in zero or more steps;
- Logic about concurrency: $[f(a,b)]_E$ rewrites to $[f(a',b')]_E$ in one step if $[a]_E$ rewrites to $[a']_E$ in one step, and $[b]_E$ rewrites to $[b']_E$ in one step;

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, how components should be linked, and how state variables should be bound to each other;

Mapping CBabel to RWL

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

components	$\longrightarrow$	classes
component's instances	$\longrightarrow$	objects
port declarations	$\longrightarrow$	messages
port stimulus	$\longrightarrow$	message passing

- The rewriting semantics that we have given to CBabel:
  - uses the object-oriented notation for rewriting logic;
  - is implemented as a transformation function in Maude's using meta-programming capabilities;

Mapping CBabel to RWL: components

Components are mapped to rewrite theories:

- Each component gives rise to a class declaration and a constructor operator;
- Component instance is represented by an object instance of such class;
- Variables are mapped to class attributes;

Mapping CBabel to RWL: ports

- Input ports and output ports could be synchronous or asynchronous;
- Ports are mapped to message declarations and port stimulus is represented as passing a message to the appropriate object;
- Port declaration *in modules* gives rise to one or two rules in rewrite theory;

Mapping CBabel to RWL: contracts

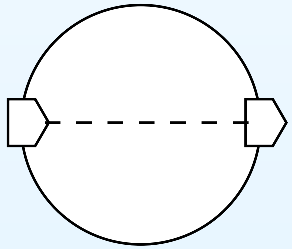
A coordination contract is a specification of the interaction flow inside a connector.

Mapping CBabel to RWL: contracts

A coordination contract is a specification of the interaction flow inside a connector.

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

A *sequential* contract specifies a “short-circuit” between a synchronous input port (i) and a synchronous output port (o).



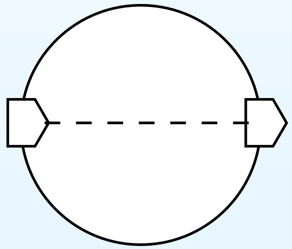
Mapping CBabel to RWL: contracts

A coordination contract is a specification of the interaction flow inside a connector.

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

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

A *sequential* contract specifies a “short-circuit” between a synchronous input port (i) and a synchronous output port (o).

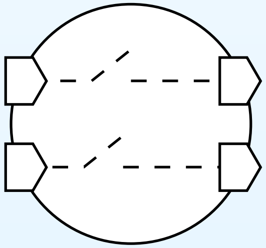


Mapping CBabel to RWL: contracts

A coordination contract is a specification of the interaction flow inside a connector.

$$send(\omega, i_n, \iota) < \omega : n \mid status : unlocked, A >$$
$$\Rightarrow < \omega : n \mid status : locked, A > send(\omega, o_n, [\omega, o_n] :: \iota)$$

A *mutual exclusive* contract between synchronous input ports has a semaphore semantics.

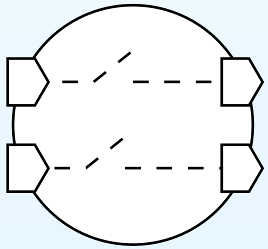


Mapping CBabel to RWL: contracts

A coordination contract is a specification of the interaction flow inside a connector.

$$
\begin{aligned}
& send(\omega, i_n, \iota) < \omega : n \mid status : unlocked, A > \\
& \qquad\qquad \Rightarrow\ < \omega : n \mid status : locked, A > send(\omega, o_n, [\omega, o_n] :: \iota) \\
& ack([\omega, o] :: \iota) < \omega : n \mid status : locked, A > \\
& \qquad\qquad \Rightarrow\ < \omega : n \mid status : unlocked, A > ack(\iota)
\end{aligned}
$$

A *mutual exclusive* contract between synchronous input ports has a semaphore semantics.

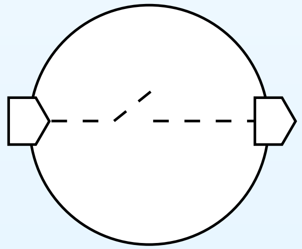


Mapping CBabel to RWL: contracts

A coordination contract is a specification of the interaction flow inside a connector.

$$send(\omega, i, \iota) < \omega : n \mid status : unlocked,\ A > \Rightarrow$$
$$before(< \omega : n \mid status : locked,\ A >)\ send(\omega, o, [\omega, o] :: \iota)\ if\ \ opened?(\ldots)$$

A *guarded sequential* contract is declared relating synchronous input and output ports. It has a condition, a before block and an after block.

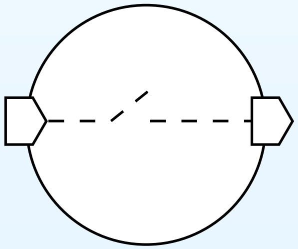


Mapping CBabel to RWL: contracts

A coordination contract is a specification of the interaction flow inside a connector.

$$send(\omega, i, \iota) < \omega : n \mid status : unlocked,\ A > \Rightarrow$$
$$before(< \omega : n \mid status : locked,\ A >)\ send(\omega, o, [\omega, o] :: \iota)\ if\ opened?(\ldots)$$
$$ack([\omega, o] :: \iota) < \omega : n \mid status : locked,\ A >$$
$$\Rightarrow\ after(< \omega : n \mid status : unlocked,\ A >)\ ack(\iota)$$

A *guarded sequential* contract is declared relating synchronous input and output ports. It has a condition, a before block and an after block.



Mapping CBabel to RWL: application

- The CBabel *application* module declares how the components of an architecture should be put together;
- Each link declaration gives rise to a rule that rewrites a message to an output port to a message to an input port.
- State required variables allow for a shared memory model of communication between CBabel components;

CBabel Tool

- The *CBabel Tool* is a direct implementation of the rewriting semantics of CBabel which allows the execution and verification of CBabel descriptions;
- It extends the *Full Maude* environment allowing one to direct import CBabel description;
- The Full Maude enviroment extends Maude with notation for object-oriented programming, parameterized modules, views and modules expressions;
- Given a *CBabel component description* and the *rewriting logic semantics* presented previously, CBabel Tool produces an Maude object-oriented module;

Executing CBabel Tool

```
$ maude cbabel-tool.maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.1.1 built: Jun 15 2004 12:55:31
             Copyright 1997-2004 SRI International
                   Wed Nov  3 21:39:20 2004

               CBabel Tool 2.0 (October 25th, 2004)

Introduced module CBABEL-CONFIGURATION

Maude>
```

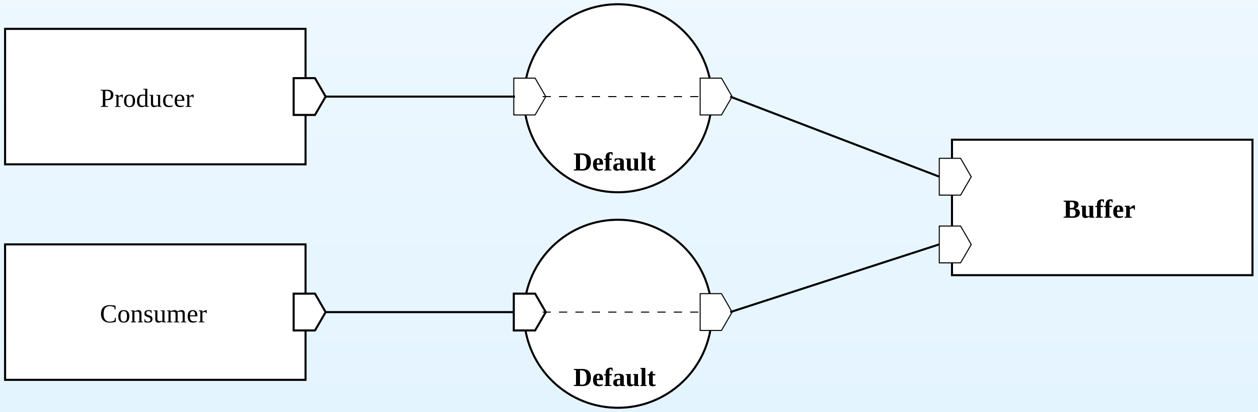
The producer-consumer-buffer example

- 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:
  - The producer and the consumer should not access the buffer at the same time, so called *race condition*;
  - If the buffer is bounded 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.

CBabel PCB-Default architecture

```
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 ;
}

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

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



PCB-Mutex and PCB-Mutex-Guards architectures

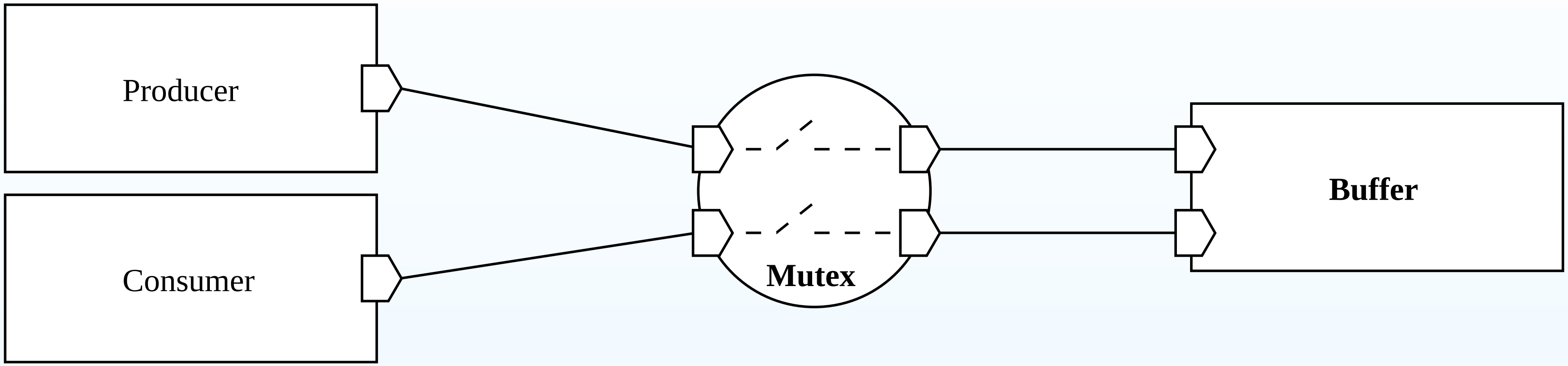


Figure 1: PCB-Mutex Architecture

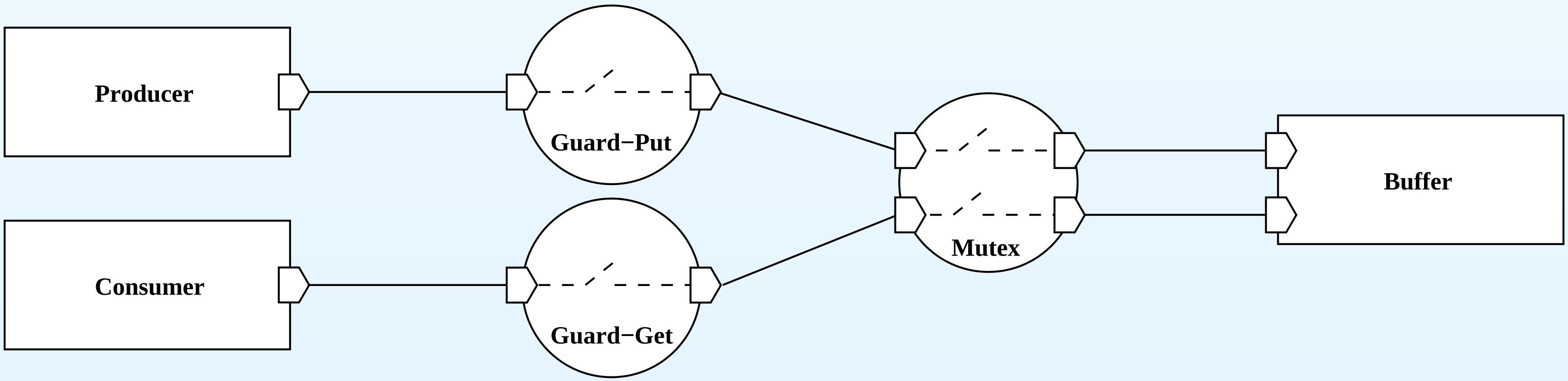


Figure 2: PCB-Mutex-Guard Architecture

The MUTEX rewriting theory

```
Maude> (show module MUTEX .)
omod MUTEX is
  including CBABEL-CONFIGURATION .

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

class and constructor

```
  op mutex-in1 : -> PortInId   [ctor] .
  op mutex-out1 : -> PortOutId [ctor] .
  op mutex-in2 : -> PortInId   [ctor] .
  op mutex-out2 : -> PortOutId [ctor] .
```

ports declaration

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

mutual exclusive contract

PCB-Mutex-Guards application rewrite theory

```
omod PC-MUTEX-GUARDS is
 including CBABEL-CONFIGURATION .
 including GUARD-PUT . including GUARD-GET .
 including BUFFER .    including PRODUCER .
 [...]
 ops cons buff gget gput prod mutx : -> Oid .
 [...]
 op topology : -> Configuration .
 eq topology = instantiate(gput,GUARD-PUT) instantiate(mutx, MUTEX)
               instantiate(prod, PRODUCER) [ ...] .
  eq < gget : GUARD-GET | items : st(V1:Int,changed) >
     < buff : BUFFER | items : V2:Int > =
     < gget : GUARD-GET | items : st(V1:Int,unchanged) >
     < buff : BUFFER | items : V1:Int > .
  ceq < gget : GUARD-GET | items : st(V1:Int,unchanged) >
      < buff : BUFFER | items : V2:Int > =
      < gget : GUARD-GET | items : st(V2:Int,unchanged) >
      < buff : BUFFER | items : V2:Int > if V1:Int =/= V2:Int = true .
  [...]
  rl send(cons, consumer-get, IT) =>  send(mutx, mutex-in2, IT)
   [label consumer-get-linking-mutex-in2] .
  [...]
endom
```

instantiations

binds

links

The verification and execution module for PCB

```
(omod S-VER-PCB is
 inc APP . inc MODEL-CHECKER .
 op initial : -> Configuration .
 eq initial =
    topology do(cons, consumer-get, none) do(prod, producer-put, none) .
 rl done(O, producer-put, IT) => do(O, producer-put, none) .
 rl done(O, consumer-get, IT) => do(O, consumer-get, none) .
 rl [buffer-do-put] :
    do(O, buffer-put, IT) < O : BUFFER | items : N, MAXITEMS : M > =>
    done(O, buffer-put, IT)
    < O : BUFFER | MAXITEMS : M ,
      items : (if (N + 1) > (M + 1) then (M + 1) else (N + 1) fi) > .
 rl [buffer-do-get] :
    do(O, buffer-get, IT) < O : BUFFER | items : N, MAXITEMS : M > =>
    done(O, buffer-get, IT)
    < O : BUFFER | MAXITEMS : M ,
      items : (if (N - 1) < -1 then -1 else (N - 1) fi) > .
 subsort Configuration < State .
 op raceCond : -> Prop .
 eq send(O, buffer-put, IT1) send(O, buffer-get, IT2)
    C:Configuration |= raceCond = true .
endom)
```

initial state

internal behaviors

race condition

Model checking in PCB-Default

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

```
reduce in VER-PCB : modelCheck(initial, [] ~ raceCond) .
```

Model checking in PCB-Default

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

```
reduce in VER-PCB : modelCheck(initial, [] ~ raceCond) .
```

The answer is *no*, and a counter-example is producer with a path wich contains a race condition state:

```
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} ...)
```

Searches and model checking in PCB-Mutex-Guards

Using the *model checker* to find a *race condition* state:

```
reduce in VER-PCB : modelCheck(initial, [] ~ raceCond) .

result Bool: true
```

Searches and model checking in PCB-Mutex-Guards

Using the *model checker* to find a *race condition* state:

```
reduce in VER-PCB : modelCheck(initial, [] ~ raceCond) .

result Bool: true
```

Using the *search* command to find a *race condition* state:

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

No solution.
```

Searches and model checking in PCB-Mutex-Guards

Using the *model checker* to find a *race condition* state:

```
reduce in VER-PCB : modelCheck(initial, [] ~ raceCond) .

result Bool: true
```

Using the *search* command to find a *race condition* state:

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

No solution.
```

Searching for buffer *overflow* state:

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

No solution.
```

Searches and model checking in PCB-Mutex-Guards

Using the *model checker* to find a *race condition* state:

```
reduce in VER-PCB : modelCheck(initial, [] ~ raceCond) .

result Bool: true
```

Using the *search* command to find a *race condition* state:

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

No solution.
```

Searching for buffer *overflow* state:

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

No solution.
```

Searching for buffer *underflow* state:

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

No solution.
```

Turn-based game (TBG)

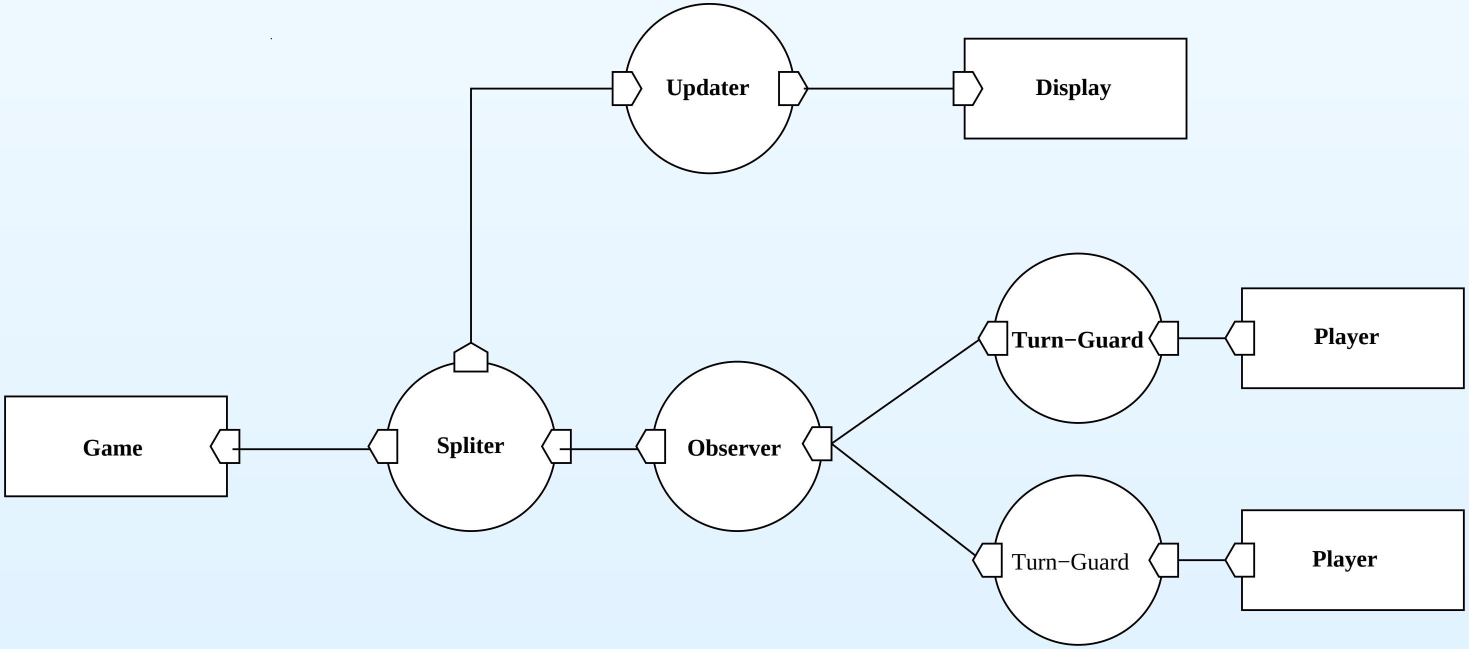
- The example illustrates how interaction *observer* design patterns can be implemented with CBabel connectors and contracts.
- Three modules: display, game and player.
- Connectors enforce the alternation of each player in the game.
- A player will only be allowed to make his turn after the previous one has been displayed and a display will only occur after a new turn is complete.

Turn-based game CBabel description

```
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 TURN-GUARD2 {
  staterequired int g2Turn ;

  in port g2TurnIn ;
  out port g2TurnOut ;

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

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

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

connector UPDATER {...}
connector OBSERVER {...}
module GAME {...}
module PLAYER {...}
application TBG {...}
```



The verification and execution module

```
(omod VER-TBG is
 inc TBG .
 inc MODEL-CHECKER .
 rl done(p1, turn, IT) => do(p1, turn, none) .
 rl done(p2, turn, IT) => do(p2, turn, none) .      players internal behavior
 rl do(display, updating, IT) => done(display, updating, IT) .
 rl do(game, gturn, IT :: [O, turn])
    < game : Game | lastPlayer : N:Int > =>
    done(game, gturn, IT :: [O, turn])
    < game : Game | lastPlayer : oidRange(O) > .    game internal behavior
 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) .     initial state

 subsort Configuration < State .
 op playing : Oid -> Prop .
 eq C < game : GAME | > send(game, gturn, IT :: [O, turn])
    |= playing(O) = true .                                         playing proposition
endom)
```

Turn-based game: verifications with search command

The alternation of the players must be preserved:

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

No solution.
```

Turn-based game: verifications with search command

The alternation of the players must be preserved:

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

No solution.
```

One turn must be finished before an update message is sent to the display:

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

No solution.
```

Turn-based game: verifications with search command

The alternation of the players must be preserved:

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

No solution.
```

One turn must be finished before an update message is sent to the display:

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

No solution.
```

A new turn must wait until the status of the former one is displayed:

```
search in VER-TBG : initial =>* C send(display, updating, IT1) send(obs,
obTurnOut, IT2) .

No solution.
```

Turn-based game: verifications with model checker

The players have always chances to make turns:

```
reduce in VER-TBG : modelCheck(initial , ([] ~ playing(p1)) /\ ([] ~
playing(p2)) ) .

result ModelCheckResult: counterexample(...
 { send(game, gturn,
[split, turnOut1] :: [obs, obTurnOut] :: [g1, g1TurnOut] ::  [p1,
turn])...}
...
 { send(game, gturn, [split, turnOut1] :: [obs,
obTurnOut] :: [g2, g2TurnOut] ::  [p2, turn])...}
...)
```

The alternation of the players is preserved:

```
reduce in VER-TBG : modelCheck(initial, [] (playing(p1) ->
                              O (~ playing(p1) R ~ playing(p2)))) .

 result Bool: true
```

Developments and future work

- Verify more complex architectural descriptions;

Developments and future work

- Verify more complex architectural descriptions;
- Execution times concerns;

Developments and future work

- Verify more complex architectural descriptions;
- Execution times concerns;
- Mapping the remaining contracts in RWL: QoS, distribution;

Developments and future work

- Verify more complex architectural descriptions;
- Execution times concerns;
- Mapping the remaining contracts in RWL: QoS, distribution;
- Complete definition of a proper command interface for the CBabel Tool that keeps the execution and verification at CBabel level;

Developments and future work

- Verify more complex architectural descriptions;
- Execution times concerns;
- Mapping the remaining contracts in RWL: QoS, distribution;
- Complete definition of a proper command interface for the CBabel Tool that keeps the execution and verification at CBabel level;
- Others improvements on the CBabel Tool (cf. architecture static verifications);

Contributions

- Since the transformation from CBabel to RWL is the actual *semantics* of CBabel, we actually *execute* CBabel to do the simulations;

Contributions

- Since the transformation from CBabel to RWL is the actual *semantics* of CBabel, we actually *execute* CBabel to do the simulations;
- Maude object-oriented syntax provides an intuitive interpretation for translated CBabel components, which is of easy understanding for most software designers;

Contributions

- Since the transformation from CBabel to RWL is the actual *semantics* of CBabel, we actually *execute* CBabel to do the simulations;
- Maude object-oriented syntax provides an intuitive interpretation for translated CBabel components, which is of easy understanding for most software designers;
- RWL provides an *orthogonal* handling of sequential aspects of the system (by equations), and its concurrent behavior (by rules);

Contributions

- Since the transformation from CBabel to RWL is the actual *semantics* of CBabel, we actually *execute* CBabel to do the simulations;
- Maude object-oriented syntax provides an intuitive interpretation for translated CBabel components, which is of easy understanding for most software designers;
- RWL provides an *orthogonal* handling of sequential aspects of the system (by equations), and its concurrent behavior (by rules);
- Adoption of Maude allows the verifications techniques to be extended in many different aspects as new improvements are added to this environment (cf. real-time features and other verification tools);