Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

Final Simulator for CML User Manual

Technical Note Number: D32.2

Version: 0.3

Date: September 2013

Public Document

http://www.compass-research.eu

Contributors:

Joey W, Coleman, AU
Anders Kaels Malmos, AU

Editors:

Joey Coleman, AU

Reviewers:

Document History

Ver	Date	Author	Description
0.1	23-08-2013	AKM	Initial document version
0.2	03-09-2013	AKM	Supported CML constructs section added
0.2	04-09-2013	AKM	Initial Cmdline section added
0.3	09-09-2013	JWC	Editing

Contents

1 The COMPASS CML Simulator

This report explains how to interpret a CML model with the COMPASS tool. This involves describing how to add and configure a launch configuration and how the interpreter is launched and used.

Before moving on, the basic modes of operation will be explained. The interpreter operates in two orthorgonal options that control its operation. The first of these is described below, and controls the level of user interaction:

1. **Simulate**: This option will interpret the model without any user interaction. When faced with an observable event choice this will be resolved by randomly picking one. The events are picked in a random but deterministic manner. Thus, the simulation will always make the same choices for every run of the same model.

2. **Animate**: This option will interpret the model with user interaction when observable events occur. When events occur the user must to choose an event before the interpretation can continue. All non-observable events will still be picked randomly.

The second option controls the interpreter's behaviour with respect to breakpoints:

1. **Run**: This will simulate/animate the model ignoring any breakpoints.

2. **Debug**: This will simulate/animate the model and pause at all enabled breakpoints.

1.1 Creating a Launch Configuration

To create a launch configuration you first click on the small arrow next to either the debug button or the run button as shown in Figure 1.

Once clicked, a drop-down menu will appear with either *Debug configurations* or *Run configurations* (depending on which button you clicked); select the appropriate *configurations* option. This will open a configurations dialog like the one shown in Figure 2.

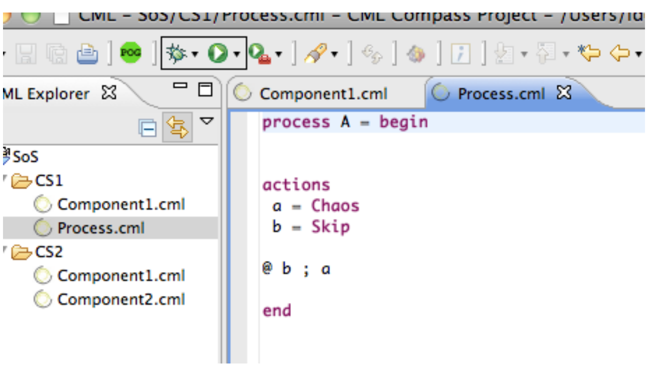

Figure 1: The debug button (the left one) and the run button (the right one), present in the toolbar at the COMPASS tool.

All of the existing CML launch configurations will appear under “CML Model”. To create a new launch configuration you must double-click on “CML Model”, then an empty launch configuration will appear as shown in Figure 2 with the name “New Configuration” (or “New configuration(<number>)” if more exists). To edit an existing one, click on the desired launch configuration name and the details will appear on the right side of the dialog.

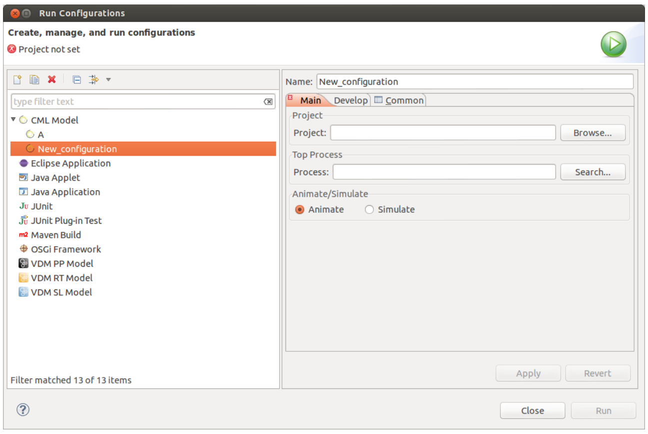

Figure 2: The launch configuration dialog showing a newly created launch configuration

As seen in Figure 2 a project name and a process name need to be associated with a launch configuration along with the mode of operation as discussed in section 1. When choosing a project, you can either write the name or click on the *browse* button which shows a list of all the available projects and choose one from there. The selection of the process name is identical.

The project name and process name must exist. It will not be possible to launch if they do not. In the left corner of Figure 2 a small red icon with an “X” and a message will indicate what is wrong. In the figure it indicates that no project has been set so this should be the first thing to do.

After setting the project name and process name, the *Apply* button must be clicked to save the changes to the launch configuration. If the project exists and is open and a process with the specified name exists in the project, then the *Run* or *Debug* button will be active and it is possible to launch the simulation as shown in Figure 3. Furthermore, the decision of whether to animate or simulate the model is decided by the two radio buttons in the buttom, the default setting is to animate.

This launch configuration will now appear in the a drop-down menu described in the beginning of this section. The actual interpretation will be described in subsection 1.3.

1.2 Launch Via Shortcut

Another way to launch a simulation is through a shortcut in the COMPASS explorer view in the CML perspective. To access this, right click on a cml file to make the

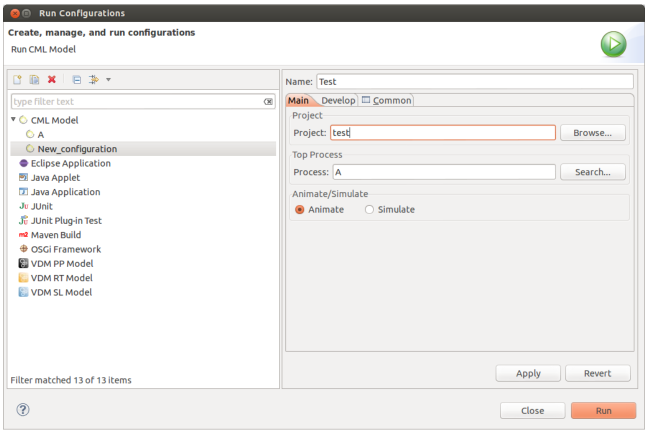

Figure 3: The *configuration dialog* after a project and process has been selected

context menu appear. From here either choose "Debug As → Debug CML Model" or "Run As → Run CML Model". After that two things can happen: if the cml file only contains one process then this process will be launched, if however more than one process is defined then a process selection dialog appears with a list of possible processes. This is shown in Figure 4.

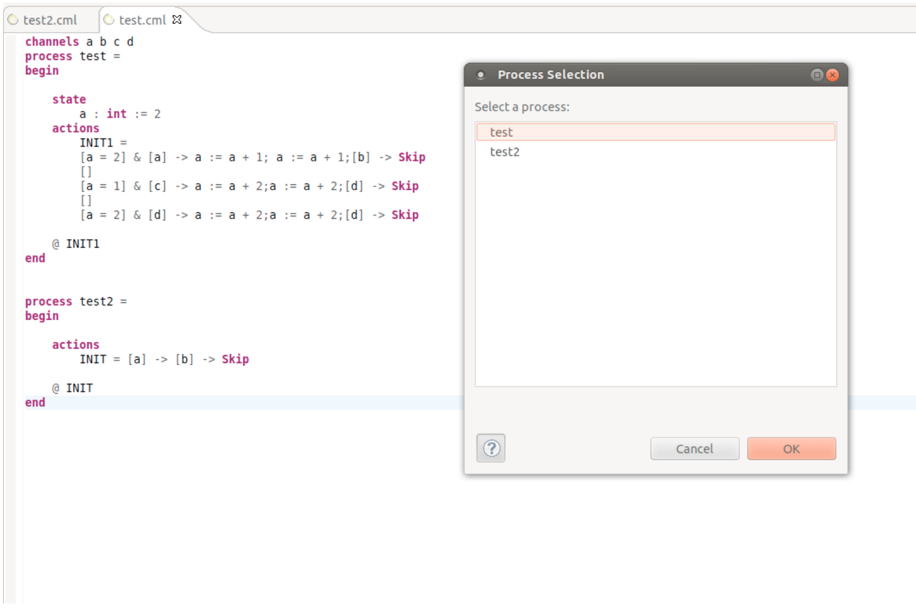

Figure 4: Right after "Run As → Run CML Model" has been clicked on, the context menu of the test.cml file appears. Since the file has defined more than one processes, the *process selection dialog* is shown.

To launch a simulation, a process must be chosen. This is done by double-clicking one of the process names in the list. This will launch a simulation with that process as the top-level process.

If you launch via a shortcut then a launch configuration named “Quick Launch” (or “Quick Launch(<number>)” if more exists) will be created and launched.

1.3 Interpretation

As mentioned at the start of section 1, there are four possible ways to interpret a model, each of them will be described.

1.3.1 Animation

Animating a model is achieved by choosing the “Animate” radio button in the launch configuration as described in the last section, this is also the default behavior. In this mode of operation the user has to pick every observable event before they can occur through the GUI.

In Figure 5 a small CML model is being animated in the debug perspective. The following windows are depicted:

Observable Event History This window is located in the top right corner and shows the observable events that has been selected so far. In the shown figure only a tock event has occurred so far.

CML Event Options This shows the possible events that can occur in the current state of the model interpretation. To make particular event occur you have to double-click it. Furthermore, to see where the process/action that offered an event is currently at, in the model, you can click it and it will be shown in the editor window.

Editor This shows the CML model source code with a twist. As seen in Figure 5 parts of the model is marked with a gray background. This marking is determined by the selected event in the CML Event Options view.

To understand how the views work together a two-step animation is shown in Figure 5 and Figure 6. In Figure 5 tock has happened ones and a tock event is currently selected. Since process A and B both offers tock they are both marked with gray in the Editor view. In Figure 6 the a event has been double-clicked and therefore just occurred. Thus, now the external choice has been resolved and only one part of the model is marked.

1.3.2 Simulating

Simulating without user interaction is achieved by choosing the “Simulate” option in the launch configuration. This mode of operation will interpret the model by taking random decisions when faced with a choice of events. However, the same choices will always be taken if the model is interpreted multiple times. In Figure 7 a simulate interpretation has completed. The most interesting view in this mode is the Observable Event History view which shows the observable trace of the interpretation.

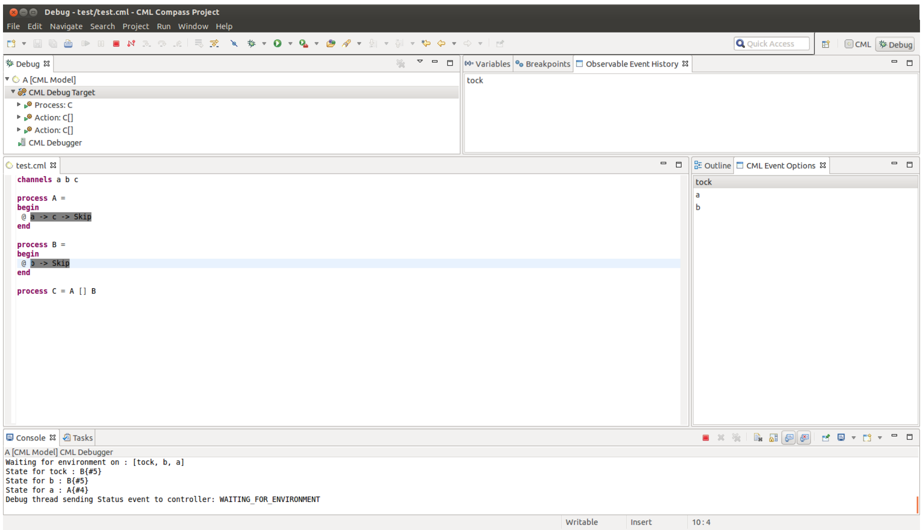

Figure 5: A CML model animated in the debug perspective.

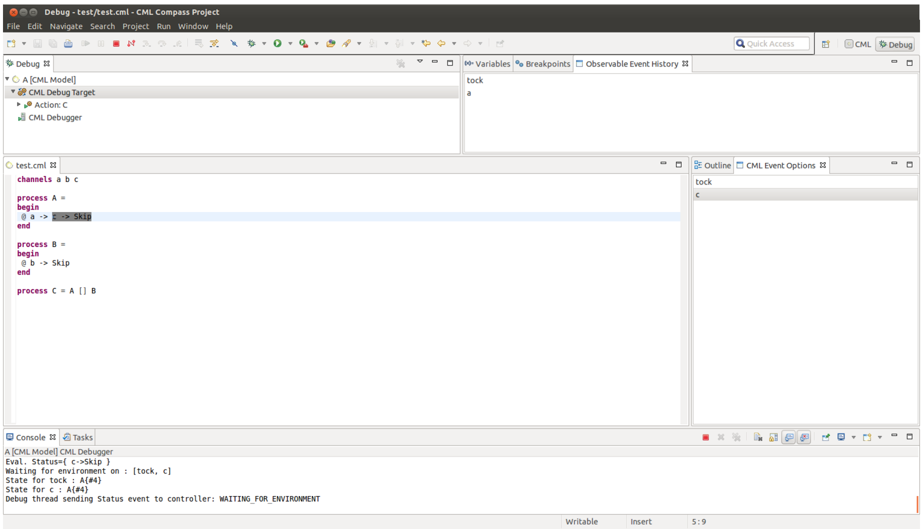

Figure 6: The a event has just occurred and the model interpretation is now currently offering the c and tock events

1.3.3 Run/Debug

In addition to the two modes of operation "Animate" and "Simulate" the standard modes "Run" and "Debug" also exists. The "Run" mode will interpret the model without ever breaking on any breakpoints. The "Debug" however will stop on any enabled breakpoint in the model.

When a "Debug" configuration is launched the perspective changes to the Eclipse Debug Perspective, "Run" will stay on whatever perspective is currently active.

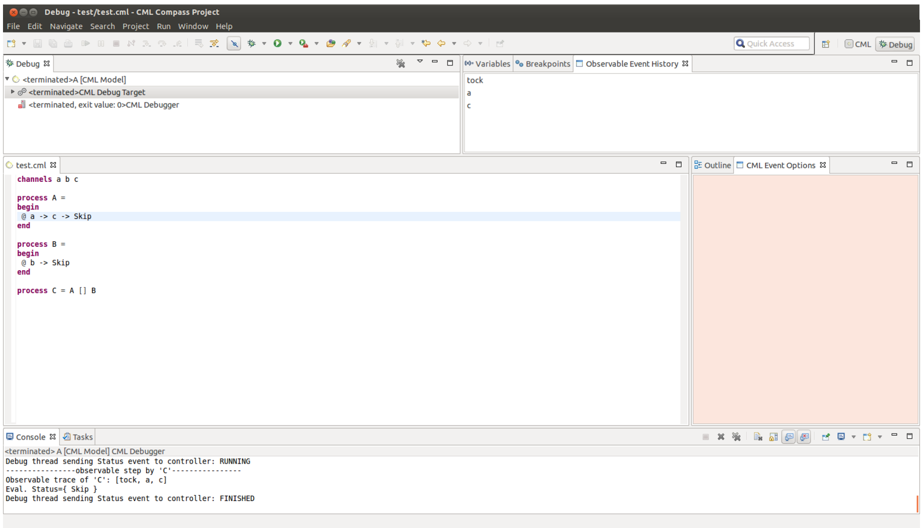

Figure 7: The model has just been simulated

To create a new breakpoint you have to double-click on the ruler to left in the editor view, if created a small dot will appear. Breakpoints can be set on processes, actions and expressions the rest will be ignored. Double-clicking on a existing breakpoint dot will remove it. In Figure 8 a debugging session is in progress. Here a breakpoint on the body of the Init method has been hit and the interpreter has been suspended. At this point the current state can be inspected in the variables view.[1] From here its both possible to resume or stop the debugging session. If the resume button is clicked the interpretation is resumed and the stop button stops it.

1.3.4 Error reporting

If an error occurs a dialog will appear with a message explaining the cause of the error. Furthermore, if possible the location of the error will be marked in the editor view. In Figure 9 a post condition has been violated. This is described in the error dialog and a gray marking shows where in the model it happend.

[1]Note: state inspection still has flaws and is not yet enabled in the development releases.

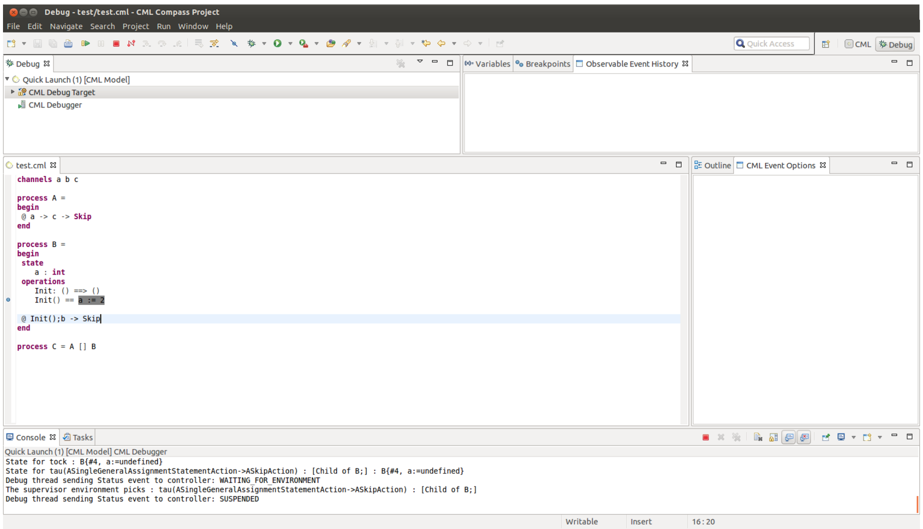

Figure 8: The interpreter is currently suspended because of a breakpoint hit marked with gray

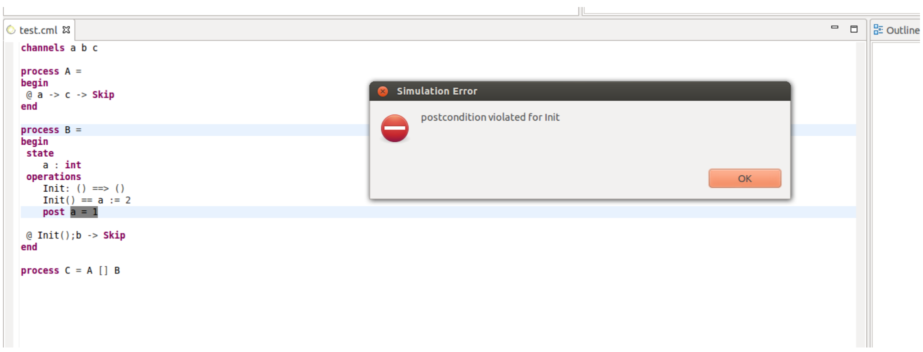

Figure 9: The interpreter has stopped because a post condition has been violated

2 Commandline Interface

The commandline tool enables simulation of CML models when invoked with the `-e` option. Since the CML model may have more than one process defined, the `-e=<processId>` option must be supplied, where `<processId>` is the name of the process that is to be simulated.

As an example of how this works, consider the following CML model in a file called `example.cml`:

```
channels
  init a b

process A = begin
  @ init -> a -> Skip
end

process B = begin
  @ init -> b -> Skip
end

process C = A;B
```

The following command will simulate the process identified by `C`:

```
cmlc -e=C example.cml
```

This results in the following output being printed to the console:

```
COMPASS command line CML Checker - CML M16
Parsing file: test.cml
1 file(s) successfully parsed. Starting analysis:
 Running The CML Type Checker on test.cml
[model types are ok]

 Running  on test.cml
Waiting for environment on : [tock, tau]
The supervisor environment picks : tock
----------------observable step by 'C'----------------------
Observable trace of 'C': [tock]
Eval. Status={ (A[]Process A)NA }
-----------------------------------------------------------------
Waiting for environment on : [tock, tau]
The supervisor environment picks : tau
Waiting for environment on : [tock, tau]
The supervisor environment picks : tau
Waiting for environment on : [tock, init]
The supervisor environment picks : tock
----------------observable step by 'C'----------------------
Observable trace of 'C': [tock, tock]
Eval. Status={ (init->a->Skip)NA }
-----------------------------------------------------------------
Waiting for environment on : [tock, init]
The supervisor environment picks : init
----------------observable step by 'C'----------------------
Observable trace of 'C': [tock, tock, init]
Eval. Status={ (a->Skip)NA }
-----------------------------------------------------------------
Waiting for environment on : [tock, a]
The supervisor environment picks : a
----------------observable step by 'C'----------------------
Observable trace of 'C': [tock, tock, init, a]
Eval. Status={ (Skip)NA }
-----------------------------------------------------------------
```

```
Waiting for environment on : [tock, tau]
The supervisor environment picks : tau
Waiting for environment on : [tock, tau]
The supervisor environment picks : tau
Waiting for environment on : [tock, tau]
The supervisor environment picks : tau
Waiting for environment on : [tock, init]
The supervisor environment picks : init
----------------observable step by ’C’----------------------
Observable trace of ’C’: [tock, tock, init, a, init]
Eval. Status={ b->Skip }
-----------------------------------------------------------------
Waiting for environment on : [tock, b]
The supervisor environment picks : b
----------------observable step by ’C’----------------------
Observable trace of ’C’: [tock, tock, init, a, init, b]
Eval. Status={ Skip }
-----------------------------------------------------------------
```

The output has these pieces of information:

Waiting for environment on: These are the events that are available for environment before the next transition is taken.

The supervisor environment picks : <event> : This shows the event that was choosen by the supervisor environment

Observable trace of ‘<processname>’: This is the top process trace, including the event that happened in this step.

Eval. Status: This is a shows the current evaluation state of the top process after the transition has been taken.

3 Supported CML Constructs

This section gives an overview of the CML constructs that are implemented. As all of the expression types are implemented, no detailed overview of them is given here.

The overview is given in two subsections: actions and statements; and processes. Each subsection contains a series of tables that group similar categories within the set of actions (and statements, processes). The first column of each table gives the name of the operator, the second gives an informal syntax, and the last is a short description that either gives the operator's status. If some operator is not implemented all or partially implemented the operator name will be red and a description of the issue will apear in the third column.

3.1 Actions

This section describes all of the supported and partially supported actions. `A` and `B` are actions, `e` is an expression, `P(x)` is a predicate expression with `x` free, and `cs` is a channel expression.

Operator	Syntax	Hints of the semantics
Termination	`Skip`	terminate immediately
Deadlock	`Stop`	
Chaos	`Chaos`	Not implemented
Divergence	`Div`	Not implemented
Delay	`Wait e`	does nothing until `e` time units have passed, and then terminates.
Prefixing	`c!e?x:P(x)-> A`	offers the environment a choice of events of the form `c.e.p`, where `p in set {x | x: T @ P(x)}`. Binding of variables presently happens after the communication, so communications such as `c?x!x` are not yet supported.
Guarded action	`[e]&A`	if `g` is true, behave like `A`, otherwise, behave like `Stop`.
Sequential composition	`A ; B`	behave like `A` until `A` terminates, then behave like `B`
External choice	`A [] B`	offer the environment the choice between `A` and `B`.
Internal choice	`A |~| B`	nondeterministically behave either like `A` or `B`.
Interrupt	`A /_\ B`	behave as `A` until `B` takes control, then behave like `B`.
Timed interrupt	`A /_e_\ B`	behave as `A` for `e` time units, then behave as `B`.
Untimed timeout	`A [_> B`	offer `A`, but may nondeterministically stop offering `A` and offer `B` at any time.
Timeout	`A [_e_> B`	offer `A` for `e` time units, then offer `B`.
Abstraction	`A \\ cs`	behave as `A` with the events in `cs` hidden
Start deadline	`A startsby e`	Not implemented
End deadline	`A endsby e`	Not implemented
Channel renaming	`A[[c<-nc]]`	Not implemented
Recursion	`mu X,Y @ ( F(X,Y), G(X,Y) )`	explicit definition of mutually recursive actions. NB: At this point mutual recursion is not implemented

Table 1: Action constructors.

<table>
<tr><th>Operator</th><th>Syntax</th><th>Hints of the semantics</th></tr>
<tr><td>Interleaving</td><td><code>A [|| ns1 | ns2 ||] B</code></td><td>execute <code>A</code> and <code>B</code> in parallel without synchronising. <code>A</code> can modify the state components in <code>ns1</code> and <code>B</code> can modify the state components in <code>ns2</code>.</td></tr>
<tr><td>Interleaving (no state)</td><td><code>A ||| B</code></td><td>execute <code>A</code> and <code>B</code> in parallel without synchronising. Neither <code>A</code> nor <code>B</code> change the state.</td></tr>
<tr><td>Synchronous parallelism</td><td><code>A [| ns1 | ns2|] B</code></td><td>execute <code>A</code> and <code>B</code> in parallel synchronising on all events. <code>A</code> can modify the state components in <code>ns1</code> and <code>B</code> can modify the state components in <code>ns2</code>.</td></tr>
<tr><td>Synchronous parallelism (no state)</td><td><code>A || B</code></td><td>execute <code>A</code> and <code>B</code> in parallel synchronising on all events. Neither <code>A</code> nor <code>B</code> change the state.</td></tr>
<tr><td>Alphabetised parallelism</td><td><code>A [ns1| X || Y |ns2] B</code></td><td>Not implemented</td></tr>
<tr><td>Alphabetised parallelism (no state)</td><td><code>A [X || Y] B</code></td><td>Not implemented</td></tr>
<tr><td>Generalised parallelism</td><td><code>A [|ns1 | cs | ns2|] B</code></td><td>execute <code>A</code> and <code>B</code> in parallel synchronising on the events in <code>cs</code>. <code>A</code> can modify the state components in <code>ns1</code> and <code>B</code> can modify the state components in <code>ns2</code>.</td></tr>
<tr><td>Generalised parallelism (no state)</td><td><code>A [| cs |] B</code></td><td>execute <code>A</code> and <code>B</code> in parallel synchronising on the events in <code>cs</code>. Neither <code>A</code> nor <code>B</code> change the state.</td></tr>
</table>

Table 2: Parallel action constructors.

Operator	Syntax	Hints of the semantics
Replicated sequential composition	`i in seq e @ A(i)`	Not implemented
Replicated external choice	`[]i in set e @ A(i)`	offer the environment the choice of all actions `A(i)` such that `i` is in the set `e`.
Replicated internal choice	`|~|i in set e @ A(i)`	nondeterministically behave as `A(i)` for any `i` in the set `e`.
Replicated interleaving	`|||i in set e @ [ns(i)]A(i)`	execute all actions `A(i)` in parallel without synchronising on any events. Each action `A(i)` can only modify the state components on `ns(i)`.
Replicated generalised parallelism	`[|cs|]i in set e @ [ns(i)] A(i)`	execute all action `A(i)` (for `i` in the set `e`) in parallel synchronising on the events in `cs`.Each action `A(i)` can only modify the state components on `ns(i)`.
Replicated alphabetised parallelism	`||i in set e @ [ns(i)| cs(i)]A(i)`	Not implemented
replicated synchronous parallelism	`||i in set e @ [ns(i)] A(i)`	execute all processes `A(i)` in parallel synchronising on all events. Each action `A(i)` can only modify the state components on `ns(i)`.

Table 3: Replicated action constructors.

Operator	Syntax	Hints of the semantics
Let	`let p=e in a`	evaluate the action `a` in the environment where `p` is associated to `e`.
Block	`(dcl v: T := e @ a)`	declare the local variable `v` of type `T` (optionally) initialised to `e` and evaluate action `a` in this context.
Assignment	`v:=e`	
Multiple assignment	`atomic (v1:=e1,...,vn:=en)`	Not implemented

Call	`obj.op(p)`
	`op(p)`
	`A(p)`
		execute operation `op` of an object `obj` (1) or of the current object or process (2) with the parameters `p`. (3) execute action `A` with parameters `p`.

Assignment call	`v := obj.op(p)` `v := op(p)`	execute operation `op` of an object `obj` (1) or of the current object or process (2) with the parameters `p` and assign the value returned by an operations to a variable.
Return	`return e` or `return`	terminates the evaluation of an operation possibly yielding a value `e`.

Specification	`[frame wr v1: T1`
	`rd v2: T2`
	`pre P1(v1,v2)`
	`post P2(v1,v1~,v2,v2~)`
	`]`
		Not implemented

New	`v := new C()`	instantiate a new object of class `C` and assign it to `v`.

Table 4: CML statements.

Operator	Syntax	Hints of the semantics

Nondeterministic if statement	`if e1 -> a1`
	`| e2 -> a2`
	`| ...`
	`end`
		evaluate all guards `ei`, if none is true, the statement diverges. If one or more guards are true, one of the associated actions is executed non-deterministically.

If statement	`if e1 then a1`
	`elseif e2 then a2`
	`...`
	`else an`
		the boolean expressions `ei` are evaluated in order. When the first `ei` is evaluated to true, the associated action is executed. If no `ei` is evaluate to true, the action in the `else` clause is executed.

Cases statement	`cases e:`
	`p1 -> a1,`
	`p2 -> a2,`
	`...,`
	`others -> an`
	`end`
		Not implemented

Nondeterministic do statement	`do e1 -> a1`
	`| e2 -> a2`
	`| ...`
	`end`
		if all guards `ei` evaluate to false, terminate. Otherwise, chose non-deterministically one guard that evaluates to true, execute the associated action, and repeat the do statement.

Sequence for loop	`for e in s do a`	for each expression `e` in the sequence `s`, execute action `a`.
Set for loop	`for all e in set S do a`	Not implemented
Index for loop	`for i=e1 to e2 by e3 do a`	Not implemented
While loop	`while e do a`	execute action `a` while the boolean expression `e` evaluates to true.

Table 5: Control statements.

3.2 Processes

This section describes all the supported and partially supported processes. `A` and `B` are both processes, `e` is an expression and `cs` is a channel expression.

<table>
<tr><th>Operator</th><th>Syntax</th><th>Hints of the semantics</th></tr>
<tr><td>Sequential composition</td><td><code>A ; B</code></td><td>behave like A until A terminates, then behave like B</td></tr>
<tr><td>External choice</td><td><code>A [] B</code></td><td>offer the environment the choice between A and B.</td></tr>
<tr><td>Internal choice</td><td><code>A |~| B</code></td><td>nondeterministically behave either like A or B.</td></tr>
<tr><td>Generalised parallelism</td><td><code>A [| cs |] B</code></td><td>execute A and B in parallel synchronising on the events in cs.</td></tr>
<tr><td>Alphabetised parallelism</td><td><code>A [X || Y] B</code></td><td>Not implemented</td></tr>
<tr><td>Synchronous parallelism</td><td><code>A || B</code></td><td>execute A and B in parallel synchronising on all events.</td></tr>
<tr><td>Interleaving</td><td><code>A ||| B</code></td><td>execute A and B in parallel without synchronising.</td></tr>
<tr><td>Interrupt</td><td><code>A /_\ B</code></td><td>Not implemented</td></tr>
<tr><td>Timed interrupt</td><td><code>A /_e_\ B</code></td><td>Not implemented</td></tr>
<tr><td>Untimed timeout</td><td><code>A [_> B</code></td><td>Not implemented</td></tr>
<tr><td>Timeout</td><td><code>A [_e_> B</code></td><td>Not implemented</td></tr>
<tr><td>Abstraction</td><td><code>A \\ cs</code></td><td>behave as A with the events in cs hidden</td></tr>
<tr><td>Start deadline</td><td><code>A startsby e</code></td><td>Not implemented</td></tr>
<tr><td>End deadline</td><td><code>A endsby e</code></td><td>Not implemented</td></tr>
<tr><td>Process instantiation</td><td><code>(v:T @ A)(e)</code> or <code>A(e)</code></td><td>behaves as A where the formal parameters (v) are instantiated to e.</td></tr>
<tr><td>Channel renaming</td><td><code>A[[c<-nc]]</code></td><td>Not implemented</td></tr>
</table>

Table 6: Process constructors.

<table>
<tr><th>Operator</th><th>Syntax</th><th>Hints of the semantics</th></tr>
<tr><td>Replicated sequential composition</td><td><code>;i in seq e @ A(i)</code></td><td>Not implemented</td></tr>
<tr><td>Replicated external choice</td><td><code>[]i in set e @ A(i)</code></td><td>Not implemented</td></tr>
<tr><td>Replicated internal choice</td><td><code>|~|i in set e @ A(i)</code></td><td>Not implemented</td></tr>
<tr><td>Replicated generalised parallelism</td><td><code>[|cs|]i in set e @ A(i)</code></td><td>Not implemented</td></tr>
<tr><td>Replicated alphabetised parallelism</td><td><code>||i in set e@[cs(i)]A(i)</code></td><td>Not implemented</td></tr>
<tr><td>Replicated synchronous parallelism</td><td><code>||i in set e @ A(i)</code></td><td>Not implemented</td></tr>
<tr><td>Replicated interleaving</td><td><code>|||i in set e @ A(i)</code></td><td>Not implemented</td></tr>
</table>

Table 7: Replicated process constructors.

4 Conclusion

At end of Month 24 the CML Simulator is not yet in its final form, but it is nearly complete. section 3 provides a comprehensive list of the constructs that are supported, and those that remain to be implemented. As it stands now, the simulator has been used in the Bang & Olufsen case study (Theme 4, WP42) and this has guided the prioritisation of implementation of operators. The remaining unimplmented operators are expected to be complete for the third release of the tool in Month 32.