Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

Final Simulator for CML User Manual

Technical Note Number: D32.2

Version: 0.6

Date: September 2013

Public Document

http://www.compass-research.eu

Contributors:

Joey W, Coleman, AU
Anders Kaels Malmos, AU

Editors:

Joey Coleman, AU

Reviewers:

Marcel Oliveira, UFPE
Ken Pierce, Newcastle
Simon Foster, York

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
0.3	04-09-2013	AKM	Initial Cmdline section added
0.4	09-09-2013	JWC	Editing
0.5	23-09-2013	AKM	Editing based on internal review
0.6	25-09-2013	JWC	Final Editing

Contents

1 The COMPASS CML Simulator

This report explains how to simulate/animate a CML model with the COMPASS tool. The tool can be downloaded from SourceForge at:

`http://sf.net/projects/compassresearch/files/Releases/0.2.0/`

where versions are available for several operating systems. This document describes how to add and configure launch configurations, and how the interpreter is launched and used.

First, the basic modes of operation are explained. The interpreter operates in two modes, "Run" and "Debug", and within these modes there are two options, "Simulate" and "Animate". The options control the level of user interaction and are described below:

1. **Simulate**: This option will interpret the model without any user interaction. When faced with a choice of several observable events, one will be chosen in a random but deterministic manner. Thus, the simulation will always make the same choices for every run of the same model. This behavior is implemented by the use of a pseudo-random number generator that has been initialised with a constant seed. This random number generator is used to resolve the choice between events, and will produce the same series of actions when presented with the same series of choices.
2. **Animate**: This option will interpret the model with user interaction. All observable events are selected by the user.

The modes of operation 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 suspend execution 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(depending on the desired mode) 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. All of the existing CML launch configurations will appear under "CML Model". To create a new launch configuration you may double-click on "CML Model" or on the "New launch configuration" button, then an empty launch configuration will appear as shown in Figure 2 with the name "New Configuration" (possibly followed by a number if this name is already used). To edit an existing configuration, click on the desired launch configuration name and the details will appear on the right side of the dialogue.

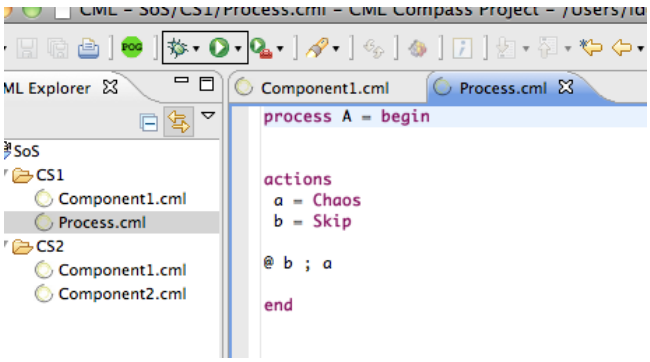

Figure 1: Screenshot of the toolbar of the COMPASS tool showing the debug button (left) and run button (right) highlighted.

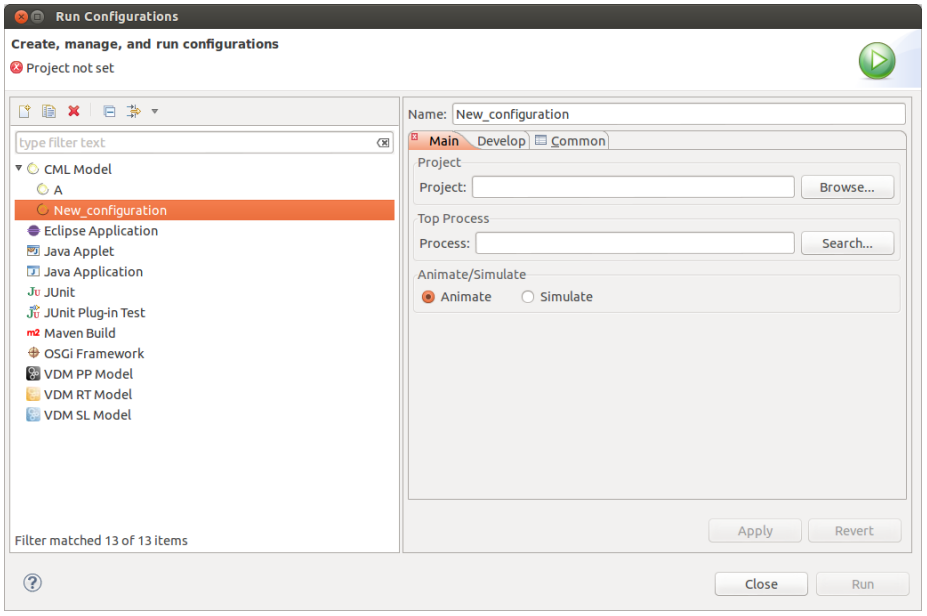

Figure 2: The launch configuration dialog showing a newly created launch configuration

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 selected project must exist in the workspace, and the process named must exist within it. 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, 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.

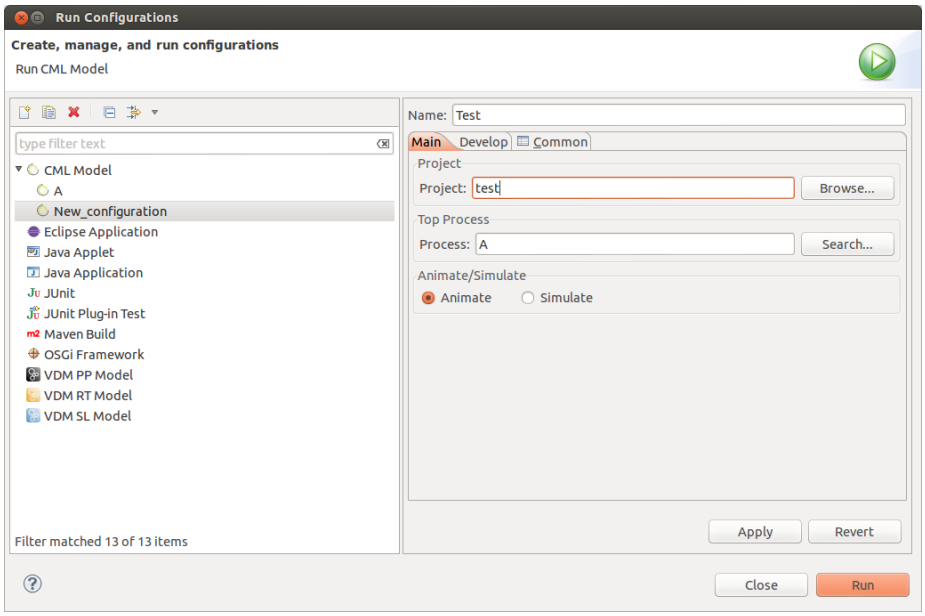

Figure 3: The *configuration dialog* after a project and process has been selected

This launch configuration will now appear in the drop-down menu as described at 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 context menu appear. From here either choose “Debug As → CML Model” or “Run As → CML Model”. After that, two things can happen: if the CML source 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.

To launch a simulation, a process must be chosen. This is done by double-clicking one of the process names in the list, or selecting it and pressing ”OK”. 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 exist) will be created and launched.

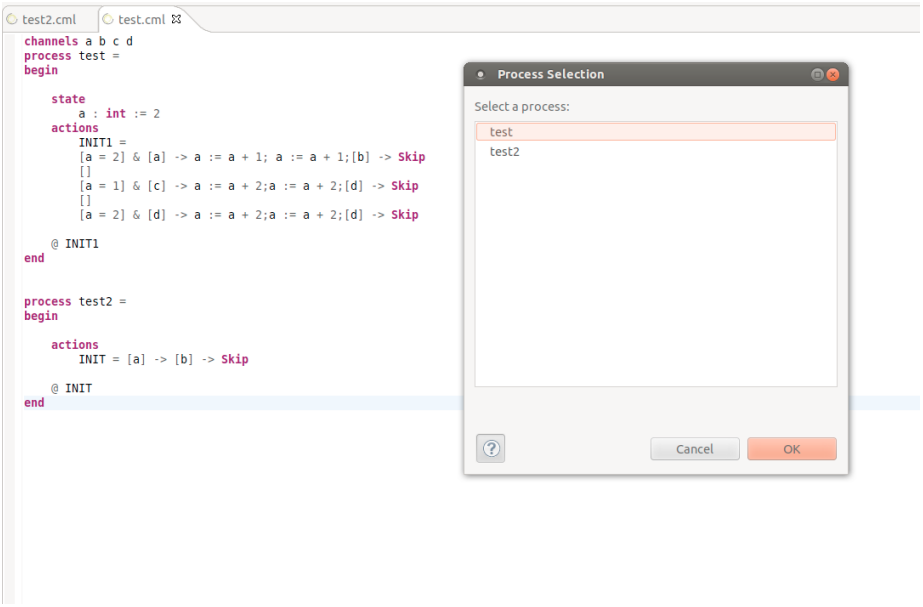

Figure 4: Right after "Run As → CML Model" has been clicked, the context menu of the test.cml file appears. Since the file defines more than one process, the *process selection dialog* is shown.

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 have been selected so far. In Figure 5 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. To make a particular event occur you must double-click it. Furthermore, to see the origin of a particular offered event, you must click it and the location of every involved construct will be marked gray 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

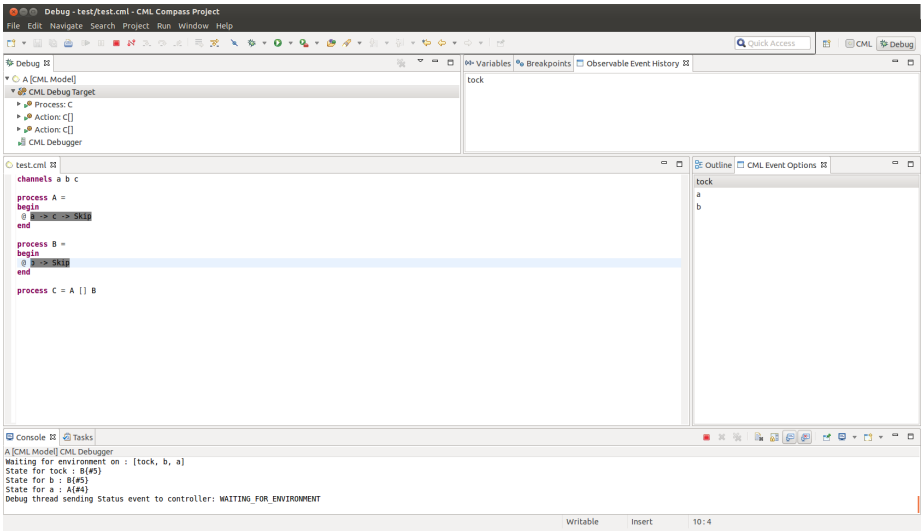

Figure 5: A CML model animated in the debug perspective.

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 once and a *tock* event is currently selected. Since process A and B both offer *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.

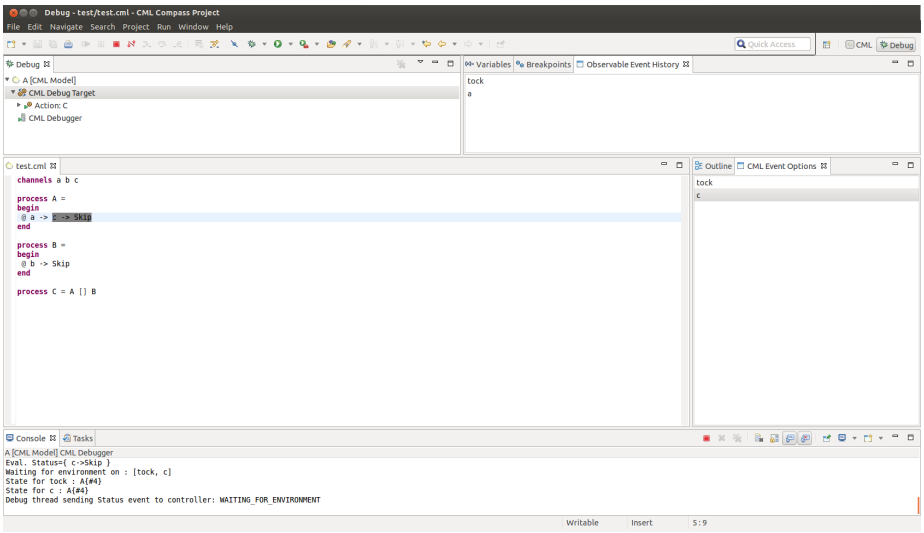

Figure 6: The a event has just occurred and the model interpretation is now currently offering *c* and *tock*

1.3.2 Simulation

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.

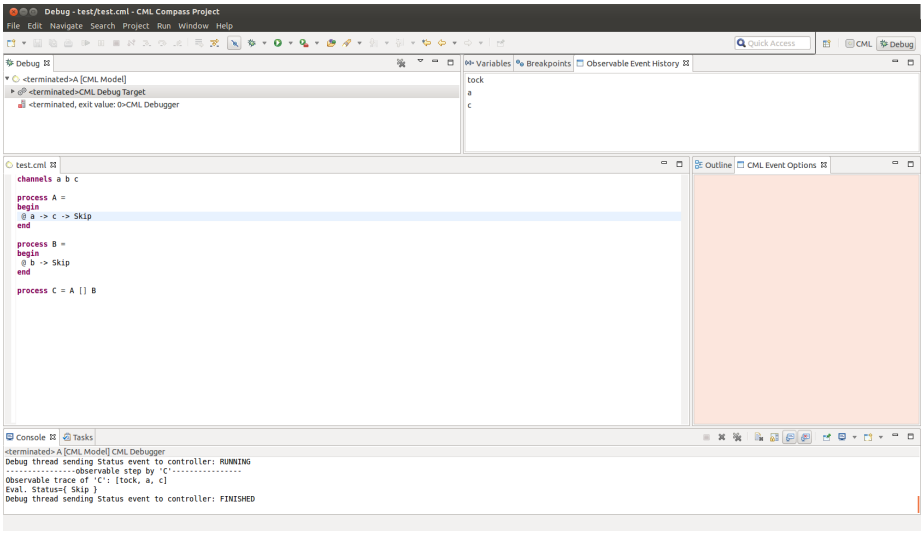

Figure 7: The model has just been simulated

1.3.3 Run/Debug

In addition to the two modes of operation "Animate" and "Simulate" the standard modes "Run" and "Debug" also exist. 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, however "Run" will stay on the perspective that is currently active.

To create a new breakpoint you have to double-click on the ruler to left in the editor view, if created, this will insert a small dot to the left ruler. Breakpoints can be set on processes, actions and expressions only. 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 it is 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] Note: state inspection still has flaws and is not yet enabled in the development releases.

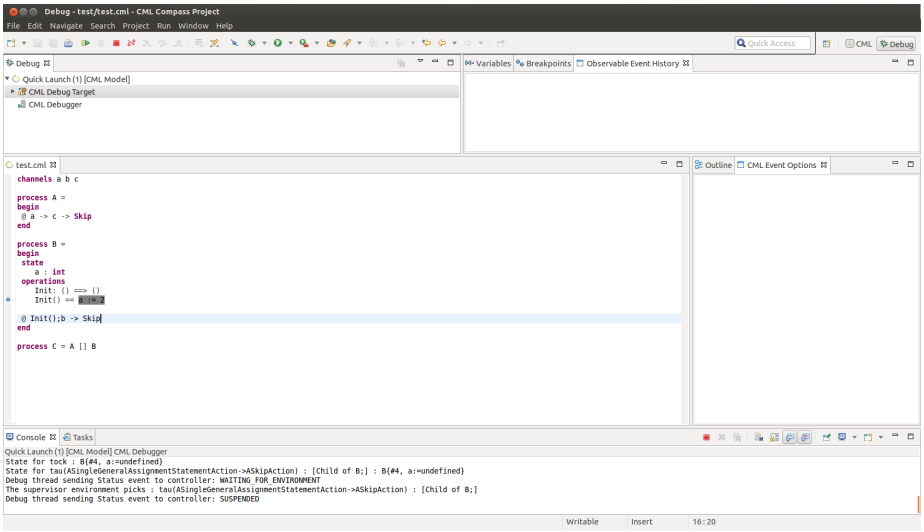

Figure 8: The interpreter is currently suspended because of a breakpoint hit marked with gray

1.3.4 Error reporting

If an error occurs a dialog will appear with a message explaining the cause of the error. Furthermore, 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 happened.

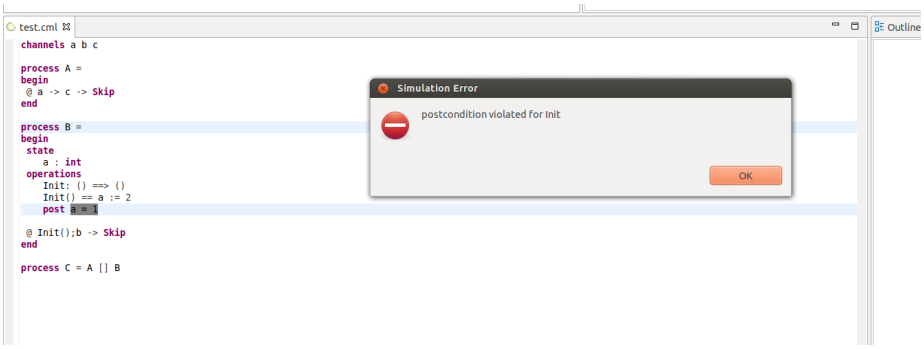

Figure 9: The interpreter has stopped because a post condition has been violated

2 Command-line Interface

The command-line interface enables animation 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 animated. If an unsupported CML construct is encountered the tool will raise an exception specifying the unsupported construct.

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 animate 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 M24
Parsing file: example.cml
1 file(s) successfully parsed. Starting analysis:
 Running The CML Type Checker on example.cml
[model types are ok]

 Running  on example.cml
Waiting for environment on : [tau(AReferenceProcess->AActionProcess) : [C;]]
The system picked: tau(AReferenceProcess->AActionProcess) : [C;]
Waiting for environment on : [tau(AActionProcess->ACommunicationAction) : [C;]]
The system picked: tau(AActionProcess->ACommunicationAction) : [C;]
Waiting for environment on : [tock, init]
[0]tock
[1]init
1
The environment picked: init
----------------observable step by 'C'----------------------
Observable trace of 'C': [init]
Eval. Status={ (a->Skip)NA }
------------------------------------------------------------------
Waiting for environment on : [tock, a]
[0]tock
[1]a
1
The environment picked: a
----------------observable step by 'C'----------------------
Observable trace of 'C': [init, a]
Eval. Status={ (Skip)NA }
------------------------------------------------------------------
Waiting for environment on : [tau(ASequentialCompositionProcess->AReferenceProcess) : [C]]
The system picked: tau(ASequentialCompositionProcess->AReferenceProcess) : [C]
Waiting for environment on : [tau(AReferenceProcess->AActionProcess) : [C]]
The system picked: tau(AReferenceProcess->AActionProcess) : [C]
Waiting for environment on : [tau(AActionProcess->ACommunicationAction) : [C]]
```

```
The system picked: tau(AActionProcess->ACommunicationAction) : [C]
Waiting for environment on : [tock, init]
[0]tock
[1]init
1
The environment picked: init
----------------observable step by 'C'----------------------
Observable trace of 'C': [init, a, init]
Eval. Status={ b->Skip }
------------------------------------------------------------------
Waiting for environment on : [tock, b]
[0]tock
[1]b
1
The environment picked: b
----------------observable step by 'C'----------------------
Observable trace of 'C': [init, a, init, b]
Eval. Status={ Skip }
------------------------------------------------------------------
Terminated with following state: FINISHED
```

The output contains the following information:

Waiting for environment on: These are the events that are available to the environment before the next transition is taken. If they include any observable events, then the interpreter will await for user input. The available events are listed together with an increasing number to the left of it. To pick an event, the user must enter the number to the left of the desired event and hit "Enter" and the animation will continue.

The environment picks : <event> : This shows the event that was chosen by the environment (e.g the user)

The system picks : <event> : This shows the silent transitions taken automatically by the interpreter.

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

Eval. Status: This shows the current state of the top-level 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 divided into two subsections: actions (and statements which is a subgroup of actions); and processes. Each subsection contains a series of tables that group similar categories. 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 gives the operator's status. If a construct is not supported entirely (no or partial implementation of the semantics), then the name of operator will be highlighted in red and a description of the issue will appear in the third column.

3.1 Actions

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

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.

Operator **Syntax**	**Comments**
Termination `Skip`	terminate immediately
Deadlock `Stop`	only allows time to pass
Chaos `Chaos`	Not implemented
Divergence `Div`	runs without ever interacting in any observable event
Delay `Wait e`	does nothing for `e` time units, 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)}`. Variable binding currently happens after communication, so forms like `c?x!x` are not yet supported.
Guarded action `[e] & A`	if `e` 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`	behave as `A`, but nondeterministically change behaviour to `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 @ F(X)`	explicit definition of a recursive action.
Mutual Recursion `mu X,Y @ (F(X,Y), G(X,Y))`	Not implemented

Table 1: Action constructors.

Operator **Syntax**	**Comments**
Interleaving `A [|| ns1	ns2 ||] B`	execute `A` and `B` in parallel without synchronising. `A` can modify the state components in `ns1` and `B` can modify the state components in `ns2`.
Interleaving (no state) `A ||| B`	execute `A` and `B` in parallel without synchronising. Neither `A` nor `B` change the state.
Synchronous parallelism `A [| ns1	ns2|] B`	execute `A` and `B` in parallel, synchronising on all events. `A` can modify the state components in `ns1` and `B` can modify the state components in `ns2`.
Synchronous parallelism (no state) `A || B`	execute `A` and `B` in parallel synchronising on all events. Neither `A` nor `B` change the state.
Alphabetised parallelism `A [ns1| cs1 || cs2 |ns2] B`	Not implemented
Alphabetised parallelism (no state) `A [cs1 || cs2] B`	Not implemented
Generalised parallelism `A [|ns1	cs	ns2|] B`	execute `A` and `B` in parallel synchronising on the events in `cs`. `A` can modify the state components in `ns1` and `B` can modify the state components in `ns2`.
Generalised parallelism (no state) `A [| cs |] B`	execute `A` and `B` in parallel synchronising on the events in `cs`. Neither `A` nor `B` change the state.

Table 2: Parallel action constructors.

Operator **Syntax**	**Comments**
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 in `ns(i)`.
Replicated generalised parallelism `[|cs|] i in set e @ [ns(i)] A(i)`	execute all actions `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 in `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 in `ns(i)`.

Table 3: Replicated action constructors.

Operator **Syntax**	**Comments**
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`	assign e to v
Multiple assignment `atomic (v1 := e1, ..., vn := en)`	Not implemented

Call
`(1)obj.op(p)`
`(2)op(p)`
`(3)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
`(1)v := obj.op(p)`
`(2)v := op(p)`
	(1) execute operation `op` of an object `obj` or (2) execute operation `op` the current object or process with the parameters `p` and assign the value returned by `op` 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**	**Comments**

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

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` evaluates to true, the action `an` 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, choose nondeterministically 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.

Operator **Syntax**	**Comments**
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`.
Generalised parallelism `A [| cs |] B`	execute `A` and `B` in parallel synchronising on the events in `cs`.
Alphabetised parallelism `A [ cs1 || cs2 ] B`	Not implemented
Synchronous parallelism `A || B`	execute `A` and `B` in parallel synchronising on all events.
Interleaving `A ||| B`	execute `A` and `B` in parallel without synchronising.
Interrupt `A /_\ B`	behave as `A` until `B` takes control, then behave like `B`.
Timed interrupt `A /_ e _\ B`	Not implemented
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 (Hiding) `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
Process instantiation `(v:T @ A)(e)` or `A(e)`	behaves as `A` where the formal parameters (`v`) are instantiated to `e`.
Channel renaming `A[[ c <- nc ]]`	Not implemented

Table 6: Process constructors.

Operator **Syntax**	**Comments**
Replicated sequential composition `; i in seq e @ A(i)`	Not implemented
Replicated external choice `[] i in set e @ A(i)`	Not implemented
Replicated internal choice `|~| i in set e @ A(i)`	Not implemented
Replicated generalised parallelism `[| cs |] i in set e @ A(i)`	execute all processes `A(i)` (for `i` in the set `e`) in parallel synchronising on the events in `cs`.

Replicated alphabetised parallelism
`|| i in set e`
`@ [cs(i)] A(i)`
	Not implemented

Replicated synchronous parallelism `|| i in set e @ A(i)`	execute all processes `A(i)` in parallel synchronising on all events.
Replicated interleaving `||| i in set e @ A(i)`	execute all processes `A(i)` in parallel without synchronising on any events.

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 unimplemented operators will be fully completed for the third release of the tool in Month 32. Most of the operators will be complete by Month 26, however, as many of the process operators have a corresponding action operators, most of unimplemented statements have corresponding implementations in the Overture platform that may be reused. The time-based operators are also expected to be complete by Month 26. The remaining operators are channel renaming and mutual recursion, and they are unlikely to have been completed until the Month 32 release.