Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

Co-simulation Engine

Deliverable Number: D32.4

Version: 1.0

Date: May 2014

Public Document

http://www.compass-research.eu

Contributors:

Kenneth Lausdahl, AU
Anders Kaels Malmos, AU
Claus Ballegaard Nielsen, AU
Joey W. Coleman, AU
Klaus Kristensen, B&O

Editors:

Joey W. Coleman, AU

Reviewers:

Ken Pierce, Newcastle
André Luís Ribeiro Didier, UFPE
Samuel Canham, York

Document History

Ver	Date	Author	Description
0.1	03-02-2014	JWC	Initial structure draft
0.2	12-02-2014	AKM	Initial structure draft and diagrams of the interpreter design
0.3	21-02-2014	KEL	Added text for the user manual
0.4	26-02-2014	KEL	Added description of the internal simulator
0.5	27-02-2014	KEL	Updated description of the simulator, implementation and initial text on the co-simulation.
0.6	28-02-2014	KEL	Added draft text for the technical sections.
0.7	04-03-2014	CBN	Added draft text for the introduction and summary sections.
0.8	05-03-2014	AKM	Updated the decription of the inspection process. Along with other minor corrections
0.9	07-03-2014	JWC	General Editing
0.10	10-03-2014	JWC	Add section on state-level implementation; editing
0.11	16-03-2014	PGL	Internal AU review
0.12	14-05-2014	JWC	Edits from internal COMPASS review comments
0.13	21-05-2014	JWC	More edits from the internal review comments
1.0	21-05-2014	JWC	Ready for submission

Contents

1 Introduction

The document describes the Co-simulation Engine component in the Symphony IDE. Co-simulation is the ability to simulate models where a portion of the modelled system is actually being simulated by other tools or systems that are external to the simulator. This means that a simulator running in one instance of the Symphony IDE may be connected to another system and, as a result, simulate the whole System of Systems (SoS) in co-operation with that system.

The co-simulation engine has two main mechanisms for use:

External System The co-simulation engine allows for the incorporation of external constituent systems, such that some of the constituent systems of the SoS are described in CML and some of the constituent systems are externally running systems. The engine allows the SoS model to be simulated with the external system(s) by enabling the interaction between the CML interpreter in Symphony and the executing external system.

Remote Simulator The co-simulation engine can be used to interact directly with other Symphony IDE instances in the same way that it allows for simulation with external systems. This allows for a simulator to be run remotely, thus establishing a distributed simulation between the tool instances. The distributed simulation is accomplished by connecting the simulators of the distributed Symphony tool instances and utilizing executable CML models.

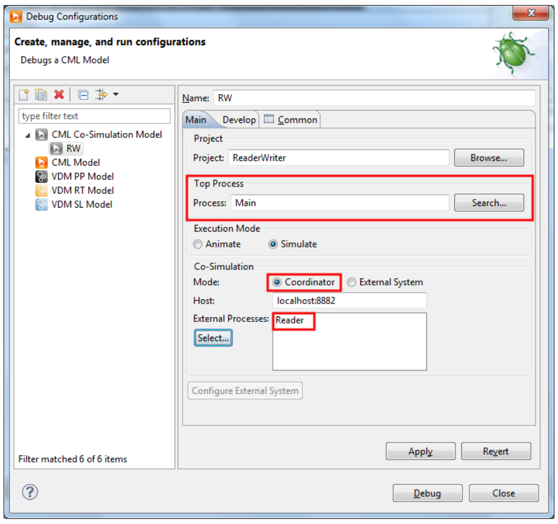
The reason for having co-simulation functionality is that the task of creating an SoS model is challenged by the autonomy and independence of the constituent system providers that participate in the SoS design. The two sources of such challenges arise from

1. the independent owners of the systems being unwilling or unable to share all knowledge of their system implementation; and,
2. the constituent systems themselves may be legacy systems with no precise description of their design, internal or external.

While the first point is social in nature, support for this sort of constraint remains important for successful SoS design. Both cases are causes of incomplete information in the design process that can only be mitigated by actually running the model against the external constituent system.

The constituent system owners in an SoS design exercise often have different agendas, and this creates a situation in which the owners may not wish to share all of the details of their systems by distributing complete models of them. Co-simulation allows the individual owners and SoS developers to perform simulations without having to share all the details of their models, as the simulation execution can be distributed across tool instances by using the co-simulation functionality. The remote simulator mechanism is used to address the first challenge source listed above.

In some cases, parts of the SoS may be delivered by a supplier that just delivers a COTS product or development version of a system, and it may be that they have no interest or obligation to deliver documentation on the internals of the systems. This makes it difficult to create CML models of all of the constituent systems making up the SoS. In other



Name: RW

Project
Project: ReaderWriter

Top Process
Process: Main

Execution Mode
( ) Animate [✓ Simulate]

Co-Simulation
Mode: [✓ Coordinator] ( ) External System
Host: localhost:8882
External Processes: Reader

Figure 1: Coordinator launch configuration with the `Reader` process as external.

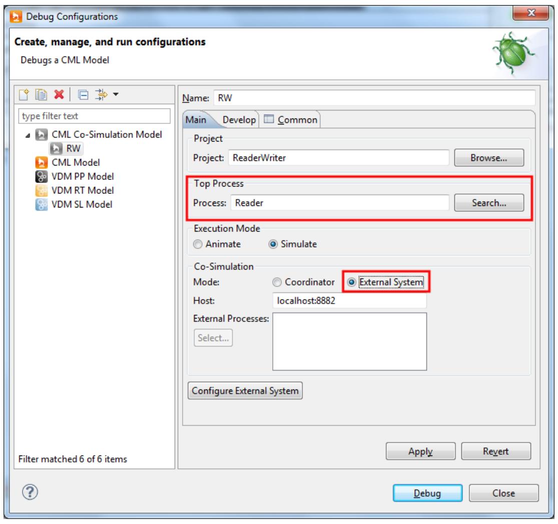
cases some of the constituent systems of the SoS may be legacy systems for which documentation of the implementation is not available. This creates a situation where the constituent systems are only described in terms of their interfaces. As a result, simulation of the CML model is a challenging task, as a meaningful simulation depends on the actual behaviour of the systems involved. In order to include the behaviour of such systems, the co-simulation engine can be used to simulate SoS models for which the behaviour of some of the constituent systems is obtained via the actual running system. The external system mechanism addresses the second challenge source above.

Both of these mechanisms build on the co-simulation engine principles, but with different configurations and different applications. The co-simulation functionality is detailed in Section 4.

The remainder of this deliverable is structured such that Section 2 provides a User's Guide to how to operate this functionality in the Symphony IDE; Section 3 gives contextual information on how the CML simulator operates; and Section 4 describes the operation of the co-simulation engine and how it modifies the operation of the base CML simulator. We conclude the document with Section 5, noting how we've trailed the co-simulation engine and some of its limitations.

2 User Manual

Co-simulation is accomplished through the use of the co-simulation configuration available under debug configurations in the Symphony IDE. A co-simulation is a distributed simulation that consists of at least one *coordinator* and any number of *external systems*. A coordinator is the simulator that simulates the toplevel CML model of the SoS. This means that it conducts its simulation at the process level, and that the simulation starts with the *top* or *entry* process, which is the process at the highest level that composes the SoS from its constituent systems.


Debug Configurations

Create, manage, and run configurations

Debugs a CML Model

type filter text

- CML Co-Simulation Model
  - RW
- CML Model
- VDM PP Model
- VDM RT Model
- VDM SL Model

Filter matched 6 of 6 items

Name: RW

Main | Develop | Common

Project

Project: ReaderWriter [Browse...]

Top Process

Process: Reader [Search...]

Execution Mode

Animate [✓ Simulate]

Co-Simulation

Mode: Coordinator [✓ External System]

Host: localhost:8882

External Processes: [Select...]

[Configure External System]

[Apply] [Revert]

[Debug] [Close]

Figure 2: External system launch configuration for the `Reader` process.

The co-simulation launch configuration shown in Figure 1 and Figure 2 allow the user to select the project, co-simulation URL, and one of two modes:

Coordinator: To run the simulator as a coordinator, the user selects the *Coordinator* option and selects the processes that will be handled by an external system. It is important to note that the process named in the *Top Process* section must be the *top* process at the SoS level. An example of such a configuration is given in Figure 1.

External System: To run the simulator as one of the constituent systems of the SoS, the user selects the *External System* option and, in this case, chooses the specific process that this simulator represents for the *Top Process* section. See Figure 2 for illustration.

Note that the options above are specific to a remote simulator configuration; an alternative configuration for external systems based on native code is described in Section 4.3.2.

A co-simulation requires the coordinator to be started first so that it is listening for clients before the external systems are started. This allows any external system to connect and register by informing the coordinator which process the external system implements. Note that current constraints mean that only one instance of a given process may be externalised: neither processes in a replicated operator nor parametrised processes may be externalised.

A partial example of a CML model is provided in Listing 1. The screenshots in Figure 1 and Figure 2 assume this example as a basis for the values in the dialogs. The example itself is a CML specification of a reader-writer system where the top SoS process is `Main`. The top process constructs the SoS using a single `Writer` and `Reader` process in parallel, synchronizing on the channels `a` and `b`. To use the co-simulation engine, the configuration must have at least one of either `Reader` or `Writer` designated as external. In Figure 1 a launch configuration is shown for the coordinator, and

```
Channels
  a : int
  b : int

process Reader = ...
process Writer = ...

process Main = Writer [|{a,b}|] Reader
```

Listing 1: A simple Reader-Writer example in CML.

Figure 2 shows the `Reader` external system. The full Reader-Writer example can be found in Appendix A.

3 Technical Overview: Simulator

This section describes how the CML simulator is structured internally and how it simulates CML specifications, though this does not include a detailed description of every language construct. The intent is to demonstrate, at a high level, how processes from a CML specification are represented in the simulator, and how they are simulated. The example shown in Listing 2 forms the basis of the explanation; it represents a small system (P) consisting of two parallel processes A and B. If time ($tock$) is ignored then only a single execution trace exists, specifically being $\langle c.1 \rangle$ where process A offers to synchronise on $c.1$ and where process B requests[1] a value on channel $c?x$.

The CML simulator uses two different approaches to simulate CML language constructs. Processes and actions are animated using an "inspect and execute" strategy, where the simulator first inspects processes and actions to construct a collection of next possible steps that could be taken. The simulator then selects a construct to execute and performs a re-inspection to select the next construct to execute. The second approach is used for all expressions which do not return the next language construct to execute, but instead execute instantaneously during inspection. This means that they are skipped, from the perspective of a step.

During simulation processes and actions are represented using an abstraction we call *behaviours*. Behaviours are derived from walking the internal simulation representation of the CML model, and are the result of inspecting and executing the language constructs. This means that every process construct is encapsulated in a behaviour, shown as circles in Figure 3. A behaviour may have children that representing nested behaviour in terms of processes or actions. A behaviour serves as a controller for its children, for whom it filters inspection results and delegates execution instructions.

An example of the system P, shown in Listing 2, is illustrated in Figure 3. It shows process P, which is a parallel composition of processes A and B synchronizing on any event on channel c. The creation and inspection behaviour is illustrated in Figure 4.

[1] $c?x$ may be thought of as an external choice of of all the possible events on channel c, so B does not strictly make a "request" to A.

```
channels
  c : nat

process P = A [|{c}|] B

process A = begin
  @ c.1 -> Skip
end

process B = begin
  @ c?x -> Skip
end
```

Listing 2: Parallel system P of A and B.

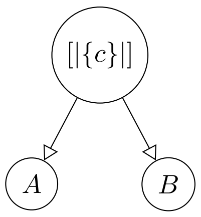


Figure 3: Internal simulator representation of P from Listing 2.

The process definition P is turned into a behaviour that is then configured and inspected to obtain possible events. When an event is selected it can be executed as shown in Figure 5. The setup and inspection shown in Figure 4 contains the following sequence of steps:

1. Creation of the behaviour from the process definition P.
2. Setup of the behaviour, possibly including the setup of child behaviors and so forth.
3. Inspection of the process. This calculates the next possible transition(s) that the behavior can make, this may include inspection of child behaviors. In this particular case the result depends on the result of its child behaviors and they are therefore inspected.
4. Inspected state. Inspection leads to an inspected state in the behaviour (represented in grey) showing the possible transitions obtained through inspection.
5. Reduction. Since the inspection in this case was performed on a alphabetised parallel composition of A and B, it must reduce the possible child transitions according to the rules of this operator. In this case the $c.1$ event is the only possible transition, so the result of the inspection of P collapses to just $c.1$.

The simulator will use the top behaviour, in this case P, to execute the selected event $c.1$. The behaviour language construct defines how to execute the event $c.1$ by passing it down to the respective children. When a child that represents an action process finishes execution the next inspect phase will change the internal language construct of that child. An action process can generally be considered complete when it reaches a state where the next language construct is a $Skip$ action, illustrated as a shaded areas in Figure 5.[2] If a behaviour has children then it will remove these based on the rules of the language construct it represents and eventually convert itself into a $Skip$ statement and thus become finished.

[2]Infinitely recursive processes never resolve to a $Skip$ action, as they never complete.

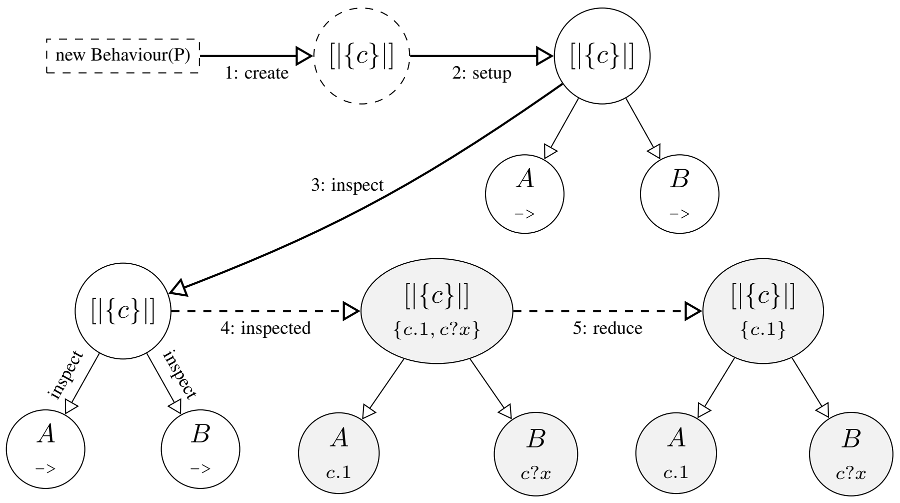


Figure 4: A simulator behaviour, showing creation and inspection.

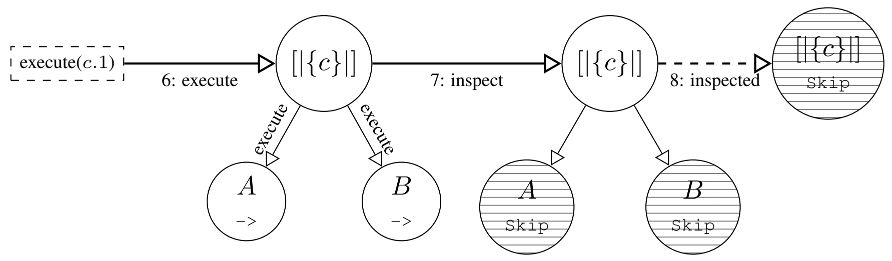


Figure 5: A simulator behaviour, showing execution and completion.

3.1 Process-level Implementation

This section gives an overview of how the simulator implements the principles shown in Figure 4 and Figure 5. The section starts by relating the elements from the figures to the equivalent static structures implemented as Java classes, then the implemented dynamic structure.

The implementation of the simulator implements the behaviour, shown as white circles in Figure 4, as a `Behaviour` class. The responsibility for the control flow is handled by the `Interpreter` class that uses an `ISelectionStrategy` to choose which event to execute if more than one event is available. It can be seen in Figure 6 that the `Behaviour` class has methods corresponding to the arrows in the previous figure. It also has fields that hold either its child processes (a left and right child for binary operators, or just a left child for unary operators) or a node that represents a language construct, such as communication (->) or `Skip`. The `Interpreter` class is responsible for implementing the overall simulation protocol that is defined by the following calls to the top behaviour: `inspect`, `isFinished`, `isDeadlocked`, and `execute`.

The `execute` operation can only be called with a chosen `Transition` and so implies that the behaviour is neither finished nor deadlocked. The events illustrated in

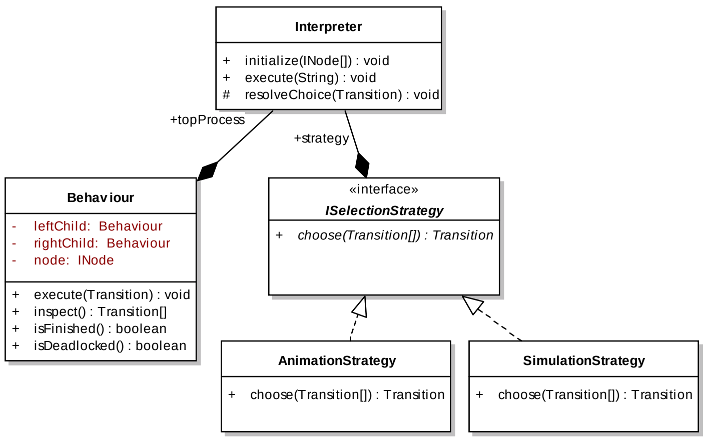

Interpreter
+ initialize(INode[]) : void
+ execute(String) : void
# resolveChoice(Transition) : void
+topProcess
+strategy
Behaviour
- leftChild: Behaviour
- rightChild: Behaviour
- node: INode
+ execute(Transition) : void
+ inspect() : Transition[]
+ isFinished() : boolean
+ isDeadlocked() : boolean
«interface»
ISelectionStrategy
+ choose(Transition[]) : Transition
AnimationStrategy
+ choose(Transition[]) : Transition
SimulationStrategy
+ choose(Transition[]) : Transition

Figure 6: Class Diagram of the core classes in the simulator.

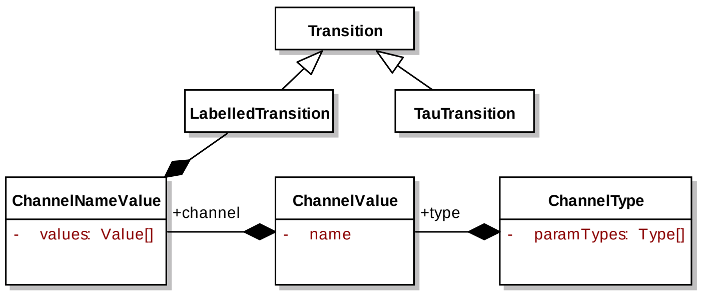


Figure 7: Class Diagram of the transition hierarchy.

brackets within the grey circles in Figure 4 are represented in the implementation by the `Transition` class shown in Figure 7. There are at least two types of transitions where the `TauTransition` class represents internal hidden transitions and the `LabelledTransition` class represents the events from Figure 4. It is the latter transitions that are of interest for co-simulation, as tau transitions are not visible outside the process in which they happen. That these transitions are labelled means that they hold the name of a channel they synchronize on, and a type that may either be void or represent a list of types from the CML language. If a channel has a non-empty list of types then an event must also contain appropriate values that match the types. These values may then be either concrete values such as the value of `1` or `<MY_TOKEN>`, or they may be a special value type that instructs the interpreter to obtain the concrete value from the user.[3]

The dynamic behaviour of the interpreter is shown in Figure 8. It is a simplified representation of the full implementation but covers the essentials. The interpretation starts by invoking the `initialize` method of the interpreter, which creates the necessary initial simulation contexts and prepares the AST represented by the specification. This

[3]The implemention for obtaining a value from the user has been omitted for clarity.

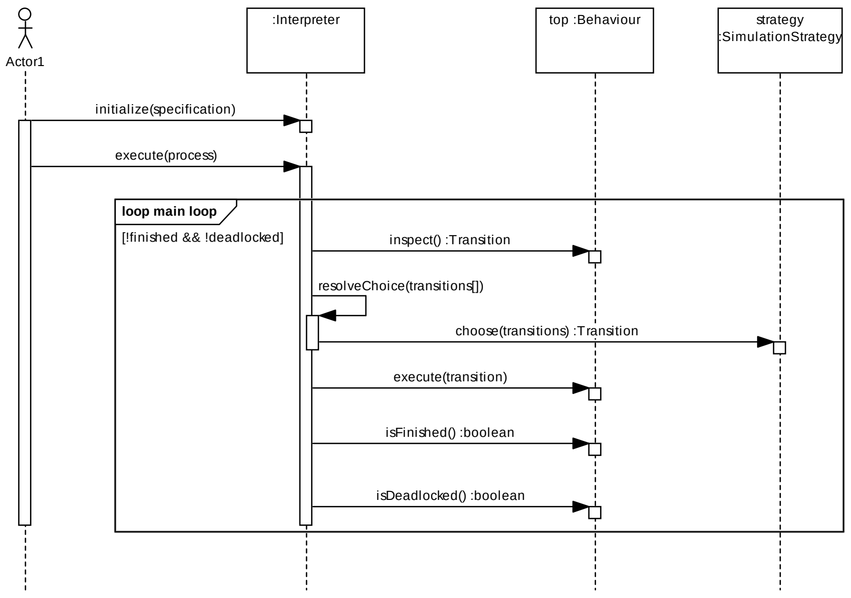


Figure 8: Sequence Diagram of the top level execution in the interpreter class.

is then followed by a call to the `execute` operation, which does a full evaluation of the specification by using a process name. This name is then matched and the process definition that matched this becomes the *top* process that will be in control of the simulation. The interpreter then enters a loop that runs as long as the behaviour does not finish or deadlock. The loop starts by inspecting the behaviour and then uses the selection strategy to select one of the possible events the behaviour may return. The event is then executed in the given context and the loop guard conditions are updated before looping.

The inspection call of a behaviour is shown in detail in Figure 9, which outlines how the behaviour uses the language constructs to obtain their children and forwards the inspection calls to them.[4]

3.2 State-level Implementation

The previous section describes the mechanism by which the simulator handles the high-level structure of a CML model. However, processes and actions within a CML model are the means of giving the model's communicative and behavioural structure, but not the details of a model's internal operation and state. This is described using the portion of CML that derives from the VDM language.

Here the simulator makes direct reuse of the Overture platform: the CML simulator incorporates the Overture VDM simulator, and any CML language construct recognised as having dervied from VDM is simply handled by the existing mechanisms in

[4]Note that the apply call to `node` has been simplified for clarity.

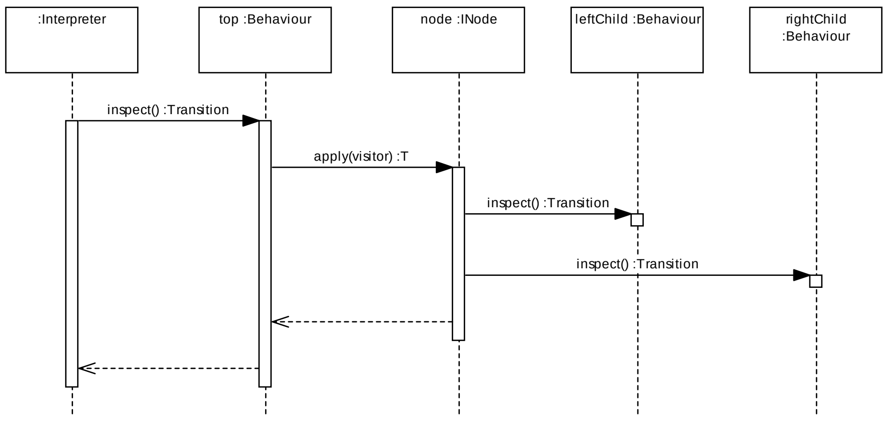


Figure 9: Sequence Diagram of the inspection of the top level behaviour.

the reused code. This includes operations, functions, almost all statements,[5] and all expressions.

However, the use of VDM constructs in CML is constrained, and some of the non-deterministic VDM constructs are not supported, in particular the non-deterministic "`let...be`" expression. This restriction means that certain patterns of modelling typically used in VDM models are, while posssible, no longer executable in a CML model, as the semantics defined in COMPASS Deliverable [BCW13] only allow a subset of VDM's constructs (including the considered removal of the "`let...be`" expression). Although it is still possible to write non-deterministic functions and operations with post conditions that require, for example, that the return value is any arbitrary member of some given set, it is no longer possible to directly execute these functions and operations.

To overcome this constraint, the Overture VDM simulator was extended to use the ProB constraint solver [LB08]. When the Overture VDM simulator encounters an implicit function or operation, it now translates the pre and post conditions into a form suitable for the ProB constraint solver and then uses the result of running ProB on it.

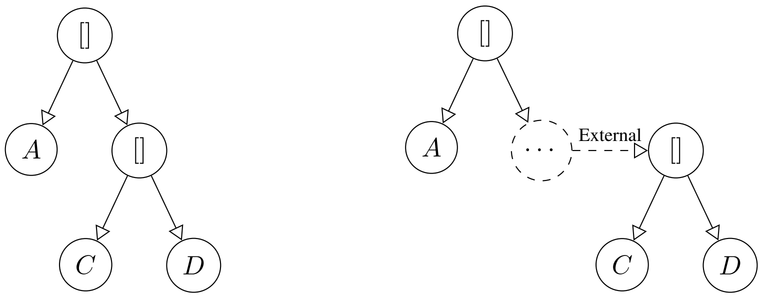
Due to the nature of the reuse that the CML simulator makes of the VDM simulator, this functionality is automatically available for CML models.

4 Technical Overview: Co-simulation

The term co-simulation will, in this context, mean the delegation of the simulation of a process to an external source. To illustrate this a small pseudo-specification is shown in Listing 3. The specification defines a top level process, P, that consists of the processes A and B, where the latter is composed of processes C and D.

```
process P = A [|{c}|] B
```

[5]CML adds non-deterministic conditional and looping constructs that are not part of VDM.



(a) Standard Simulator setup (b) Externalized Simulator setup

Figure 10: Illustration of the internal simulator behaviour setup.

```
process A = ...

process B = C [|{b}|] D

process C = ...
process D = ...
```

Listing 3: Parallel system P of A and B.

This specification can then be illustrated graphically for a standard simulator as shown in Figure 10a. However, if the process B were externalized and simulated by a remote simulator, then the tree would have to be modified as shown in Figure 10b. The solid lines in both figures represent method calls of the behaviour interface (described in Section 3.1). The dashed line represents the same interface accessed over a network connection using a common communication protocol.

In the following, Section 4.1 describes the fundamentals of the simulation network protocol, Section 4.2 describes how the simulator can be modified to enable behaviours to run externally from the simulator itself, and Section 4.3 describes how the simulator can be modified to run as an external simulator for a process, and also how external code can be connected as an implementation of an external process.

We have also provided a CML model of the simulation network protocol in Appendix B.

4.1 Network Protocol

The network protocol enables the externalization of a behaviour by allowing the calls defined in the `Behaviour` class to be transmitted over a network connection to a remote implementation of the behaviour. The underlying protocol chosen is Transmission Control Protocol (TCP) [Pos81], because of its reliability and ability to accurately deliver messages. The TCP protocol only handles streams of bytes and therefore a data transmission format must be used to allow the Java implementation of the simulator to communicate with external systems, such as software implemented on different platforms. The JavaScript Object Notation (JSON) language was chosen, specifically JSON-RPC [Cro06], to facilitate the communication of state and `Transition` values across TCP connections.

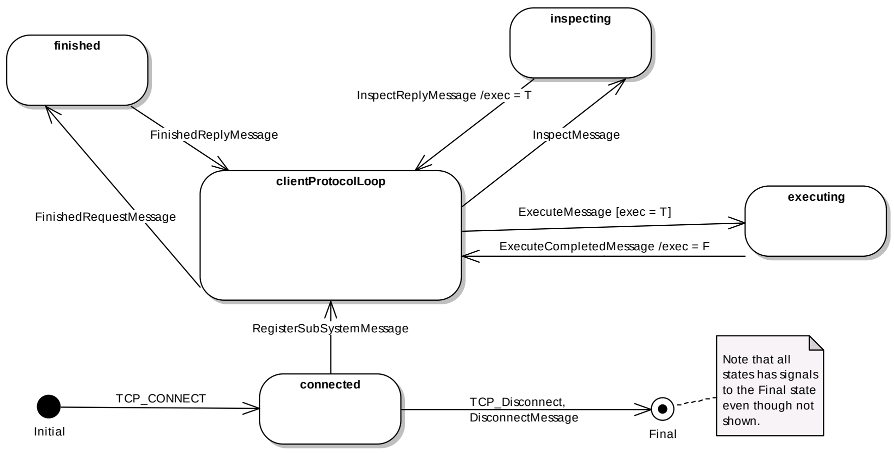

finished
inspecting
InspectReplyMessage /exec = T
FinishedReplyMessage
InspectMessage
clientProtocolLoop
executing
FinishedRequestMessage
ExecuteMessage [exec = T]
ExecuteCompletedMessage /exec = F
RegisterSubSystemMessage
Note that all states has signals to the Final state even though not shown.
connected
TCP_CONNECT
TCP_Disconnect, DisconnectMessage
Initial
Final

Figure 11: State Diagram of the external simulation protocol.

The protocol itself is described in Figure 11, showing when it is allowed to transmit various messages. The protocol complements the control flow described in Figure 8, and includes an extra control flow state *executing* that only allows an execution message if an inspection has previously been processed. The protocol also has a message to register which process the external system supplies to the coordinator before the client protocol loop starts. It must also be noted that the protocol allows a client to receive a disconnect message or disconnect itself from the TCP connection at any time.

4.2 Process Behaviour Delegation (Coordinator)

The simulator has been extended to allow processes to be externalised; however only minimal changes were required to the simulator itself. A new sub class of `Behaviour` that delegates calls to a remote client using the protocol from Section 4.1 is needed. A server setup is necessary to enable the simulator to listen on sockets and registering connecting clients such that they can be linked with the `ExternalBehaviour` instances that represents the external processes. The sequence diagram shown in Figure 12 extends Figure 9 with the external behaviour for the `rightChild` in the form of a remote simulator. The external implementation may either be a remote CML simulator as shown in the figure or a custom-developed system implementation, as long as they implement the same protocol as in Section 4.1.

4.3 Process Behaviour Contribution (Clients)

A process behaviour contribution is represented by either a simulator implementing the client version of the protocol from Section 4.1, such as the Symphony IDE, or as a custom implementation in any programming language capable of communicating using TCP. A remote simulator is a simulator that has been modified such that the control is delegated to a coordinator. Section 4.3.1 explains how the existing simulator has been modified to allow it to act as a remote simulator. The extended implementation of a

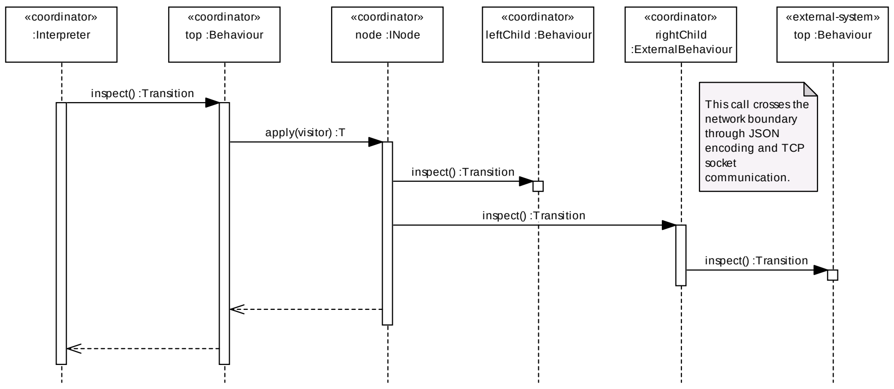

«coordinator» :Interpreter
«coordinator» top :Behaviour
«coordinator» node :INode
«coordinator» leftChild :Behaviour
«coordinator» rightChild :ExternalBehaviour
«external-system» top :Behaviour
inspect() :Transition
apply(visitor) :T
inspect() :Transition
inspect() :Transition
inspect() :Transition
This call crosses the network boundary through JSON encoding and TCP socket communication.

Figure 12: Sequence Diagram of the inspection of the top level behaviour when using external behaviours.

system (a process) may also be used for external process simulation. This is achieved by mapping CML events to state changes and method calls in the system. A short overview of this is provided in Section 4.3.2; however, for a more detailed description see [LL14].

4.3.1 Remote Simulator

The remote simulator is based on the standard simulator but has a client implementation of the protocol starting its simulation by registering its contribution in the coordinating simulator. Then it follows the execution flow shown in Figure 13. It is similar to Figure 8 but has the main execution loop changed such that it automatically executes internal silent transitions (i.e. `TauTransition`) until none are available. When all available silent transitions are executed then it calls the `Client`, fetching the available transitions from the coordinator, which only ever consists of a single transition. The simulator has another thread the handles the protocol, and it is this thread that explicitly calls `interpreter.inspect()` to supply the coordinator with inspection transitions upon request.

4.3.2 External System Implementation

An external system implementation may be an implementation of a subsystem in any programming language that is capable of TCP communication. The system must implement the protocol described in Section 4.1 in the same manner as the remote simulator. The implementation of the protocol consists of a mapping from the type representation of the concrete programming language to the types used in the simulation interface being CML types described in JSON, and a mapping from the protocol messages to the internal state machine of the external system itself.

As a result of this design, the *inspect* message must be answered with a list of transitions that represent the possible events that are allowed from the current state, and a mapping from the *execute* message to a state change of the system state. These state

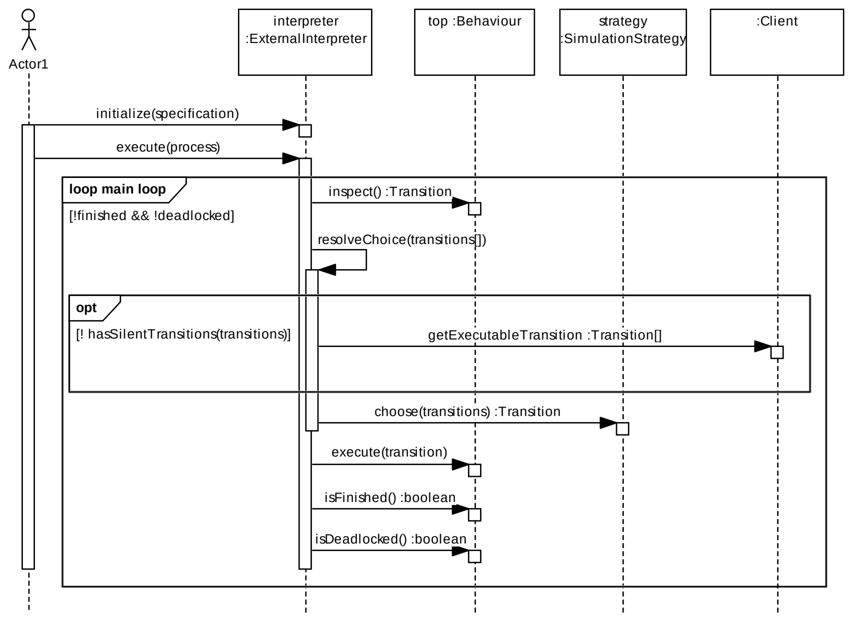

Actor1
interpreter :ExternalInterpreter
top :Behaviour
strategy :SimulationStrategy
:Client
initialize(specification)
execute(process)
loop main loop
[!finished && !deadlocked]
inspect() :Transition
resolveChoice(transitions[])
opt
[! hasSilentTransitions(transitions)]
getExecutableTransition :Transition[]
choose(transitions) :Transition
execute(transition)
isFinished() :boolean
isDeadlocked() :boolean

Figure 13: Sequence Diagram of the top level inspection for the remote simulator being connected as a client.

changes may then lead to a change of events being returned from the subsequent inspection message.

There is a subtle difference in behaviour between the CML simulator and an external implementation. The simulator, run remotely, will never report as available events that are not expected. An implementation may, however, report all possible events as available, but report errors on those that are not expected.

A concrete example of how such mappings can be achieved is described in [LL14], which also presents a framework for managing the type and event mappings.

5 Conclusion

In this document we described the operation of the co-simulation engine in the Symphony IDE which allows the simulation of CML models to be performed in co-operation with systems that are external to the current tool instance. This enables a move from having a localized model in a single tool instance to having simulator with open and versatile simulation capabilities.

The way the co-simulation engine is designed and implemented it can include externally-running systems into the simulation of local CML models, as well as distributed simulation of a CML model, divided between instances of the Symphony IDE. The key goal has been to include constituent systems that –for whatever reason– cannot be modelled in detail into the simulation of a whole CML model, and this is now possible with some limitations.

There are two limitations of the approach: the external processes must be unique by name and co-simulation does not support models that use time-based CML constructs (i.e. wait, startsby, endsby, timeout, and interrupt). The limitation on time-based constructs is caused by the difficulty of mapping the time-related constructs in the CML language, and could be overcome by a semantic mapping between the time-based constructs and their operational behaviour in real-time systems.

The co-simulation engine has been trialled in practice using a media streaming device from B&O. The deviced was embedded as an external process into a CML model of the overall system in which the streaming device would be placed, and the overall system (less the device) was simulated by the CML simulator.

References

[BCW13] Jeremy Bryans, Samuel Canham, and Jim Woodcock. CML Definition 3 — Denotational Semantics. Technical report, COMPASS Deliverable, D23.4, September 2013. Available at `http://www.compass-research.eu/`.

[Cro06] D. Crockford. The application/json media type for javascript object notation (JSON). RFC 4627, Internet Engineering Task Force, July 2006.

[LB08] M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[LL14] Kenneth Lausdahl and Adrian Larkham. Linkage to executable software. Technical report, COMPASS Deliverable, D32.3, June 2014.

[Pos81] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

A Example

This appendix provides the full source of the example outlined in Listing 1 on page 8.

```
channels
  a: int
  b: int

values
  MAX = 10

process Main = Writer [|{a,b}|] Echo

process Echo = Reader
process Reader = begin
  actions
    s = a?x -> (
          [x < MAX] & b!x -> s
          []
          [x = MAX] & b!x -> Skip
        )
  @ s
end

process Writer = begin
  actions
    s = val x : int @ a!x -> b?y ->
                        let n = y+1
                        in (
                          [n <= MAX] & s(n)
                          []
                          [n > MAX] & Skip
                        )
  @ s(1)
end
```

Listing 4: A simple Reader-Writer example in CML.

B Protocol

This appendix provides a CML of the simulation network protocol used by the co-simulation engine (as covered in Section 4).

```
types
  TCP_Event = <CONNECT> | <DISCONNECT>
  CoSimulationProtocol = <RegisterSubSystemMessage>
                       | <InspectMessage>
                       | <InspectReplyMessage>
                       | <ExecuteMessage>
                       | <ExecuteCompletedMessage>
                       | <FinishedRequestMessage>
                       | <FinishedReplyMessage>
                       | <DisconnectMessage>

  ChannelEventObject = nat
  ChannelObjectSet = set of ChannelEventObject

  String = seq of char

channels
  ch_tcp : TCP_Event | CoSimulationProtocol
  ch_init

class IO = begin
  operations
    public println : int ==> ()
    println(arg) ==
      is not yet specified

    public println : String ==> ()
    println(arg) ==
      is not yet specified

     -- need to be a static class, must sync access to the native
     -- io stream
     public toString : int ==> ()
     toString(i) ==
       is not yet specified
end

class CoSimulationCallback = begin
  state
    debug:[IO] := nil

  operations
    public CoSimulationCallback : () ==> CoSimulationCallback
    CoSimulationCallback() == (debug := new IO())

    public inspect : () ==> ChannelObjectSet
    inspect() == (
      debug.println("inspecting");
      return {}
    )

    public execute : ChannelEventObject ==> ()
    execute(-) == (
      debug.println("executing")
    )
    public finished : () ==> bool
    finished() == (
      debug.println("finished");
      return false
    )
end

process CoSimlulationServer = begin
  state
    isExecuteable : bool := false

  operations
    setExeState : bool ==> ()
    setExeState(b) == isExecuteable := b

  actions
    act_serviceLoop = ch_tcp.<CONNECT> -> (
      act_connected
      /_\
      ( ch_tcp.<DISCONNECT> -> Skip
        []
        ch_tcp.<DisconnectMessage>->Skip
      )
    )

    act_connected =
      ch_tcp.<RegisterSubSystemMessage> -> act_clientProtocolLoop

    act_clientProtocolLoop =
      ( act_inspecting
        []
        act_finished
        []
        [isExecuteable] & act_executing
      ) ; act_clientProtocolLoop

    act_inspecting = ch_tcp.<InspectMessage> ->
                     ch_tcp.<InspectReplyMessage> ->
                     (setExeState(true)) ; Skip

    act_executing = ch_tcp.<ExecuteMessage> ->
                    ch_tcp.<ExecuteCompletedMessage> ->
                    (setExeState(false)) ; Skip

    act_finished = ch_tcp.<FinishedRequestMessage> ->
                   ch_tcp.<FinishedReplyMessage> ->
                   Skip

  @ ch_init -> ( act_serviceLoop
                 /_\
                 ch_init -> Skip
               )
end

process CoSimlulationClient = begin
  state
    myCallback : [CoSimulationCallback] := nil
    isExecuteable:bool := false

  operations
    setExeState : bool ==> ()
    setExeState(b) == isExecuteable := b
    initObject : () ==> ()
    initObject() == ( myCallback := new CoSimulationCallback() )

  actions
    act_serviceLoop =
      ch_tcp.<CONNECT> -> (
        act_connected
        /_\
        ( ch_tcp.<DISCONNECT> -> Skip
          []
          ch_tcp.<DisconnectMessage> -> Skip
        )
      )

    act_connected =
      ch_tcp.<RegisterSubSystemMessage> -> act_clientProtocolLoop

    act_clientProtocolLoop = (
      act_inspecting
      []
      act_finished
      []
      [isExecuteable] & act_executing
    ) ; act_clientProtocolLoop

    act_inspecting =
      ch_tcp.<InspectMessage> -> ( myCallback.inspect() ) ;
      ch_tcp.<InspectReplyMessage> -> (setExeState(true)) ;
      Skip

    act_executing =
      ch_tcp.<ExecuteMessage> -> ( myCallback.execute(1) ) ;
      ch_tcp.<ExecuteCompletedMessage>->(setExeState(false)) ;
      Skip

    act_finished =
      ch_tcp.<FinishedRequestMessage> -> ( myCallback.finished() ) ;
      ch_tcp.<FinishedReplyMessage> -> Skip

  @ ch_init->initObject()
  ; ( act_serviceLoop
      /_\
      ch_init -> Skip
    )
end

process CoSimlulationClientAndServer =
  CoSimlulationServer [| {| ch_init, ch_tcp |} |] CoSimlulationClient
```