Chapter 13
Semantics of Co-simulation

Joey W. Coleman, Kenneth Lausdahl, and Peter Gorm Larsen

13.1 Introduction

This chapter provides an overview of the semantics for the whole co-simulation framework, covering the co-simulation engine and the scripting language. It is presented in a combination of styles, including graphical representations and Structural Operational semantics (SOS), see [77, 78]. This chapter assumes a background in formal semantics and is intended primarily for readers seeking a full understanding of the semantics underlying the Crescendo technology.

The mathematical foundations for bond graphs and the VDM-RT notation are substantially different. In this chapter, we present the formal foundations that enable coupling of these diverse formalisms to permit co-simulation. The Crescendo tool implements the semantics presented here following a traditional master/slave architecture. The master is the co-simulation engine that manages all communication between the DE and CT simulators which act as slaves and progress the simulation according to the directives coming from the master co-simulation engine.

The overall structure of the co-simulation engine, a basic explanation of the synchronisation of the two "slave" simulators, and the semantic constraints for the simulators are presented in Sect. 13.2. Section 13.3 provides the actual co-simulation semantics. Then, Sects. 13.4 and 13.5 explain how the co-simulation can be extended using the semantics for a small language Crescendo Scripting Language (CSL) that enables scenarios to be exercised by a co-simulation. Section 13.6 ends the chapter with a short summary.

J.W. Coleman (✉) • K. Lausdahl • P.G. Larsen
Aarhus University, Aarhus, Denmark
e-mail: jwc@eng.au.dk; lausdahl@eng.au.dk; pgl@eng.au.dk

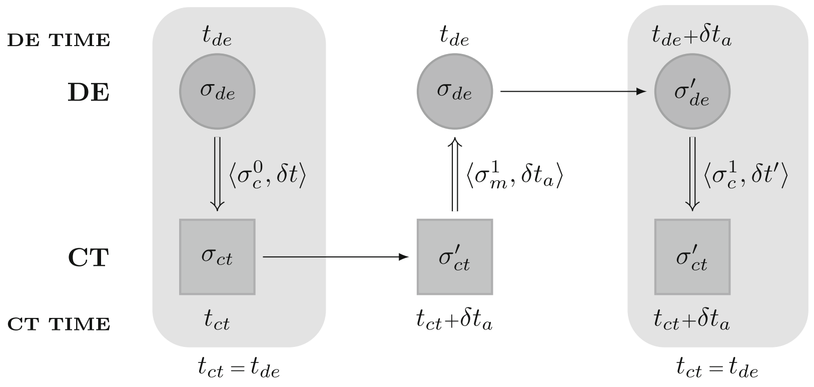

Fig. 13.1 Example of the synchronisation scheme for DE-CT co-simulation

13.2 Structure of Co-simulation

Figure 13.1 illustrates the synchronisation scheme underlying co-simulation between a DE simulation of a controller (top) and a CT simulation of the plant (bottom). It is an expanded diagram of Fig 2.3. The DE and CT simulators are coupled through a co-simulation engine that explicitly synchronises the shared variables, events and the simulation time in both linked simulators (the co-simulation engine is not shown explicitly in Fig. 13.1).

Each simulation maintains its own local state and time at which the state is valid. Thus, let σ_{de} be the internal state of the DE simulation at simulation time t_{de}, and let σ_{ct} be the internal state of the CT simulation at simulation time t_{ct}. The controlled variables defined in the co-simulation contract (whose values are defined in σ_{c}) are set by the DE controller and read by the plant; the monitored variables (σ_{m}) are set by the plant model and read by the controller.

Consider a synchronisation cycle which starts with the two simulators having a common simulation time ($t_{\mathrm{ct}} = t_{\mathrm{de}}$). On each cycle, the DE controller simulation sets the controlled variables and proposes a duration δt by which the CT simulation should, if possible, advance. As the CT simulation of the environment advances, it may encounter a state in which one of the event predicates defined in the contract becomes true. The state of the monitored variables σ_{m}^1 and the actual time that it reached δt_{a} are communicated back to the DE side. If no events occur in the CT simulation during this interval, $\delta t_{\mathrm{a}} = \delta t$. While the CT simulation has been progressing, the DE simulation remains unchanged, so its local simulation time remains at t_{de} and state σ_{de}. The DE simulation then advances by δt_{a} so that both DE and CT are again synchronised at the same simulation time, and the controlled variables are updated (σ_{c}^1) and the next time step is proposed to CT. The performance of the DE state change takes place in two stages, with the calculations being performed first, separately from advancing the DE simulation time. The granularity of the synchronisation time step is always determined by the DE simulator. The scheme does not require resource-intensive roll-back of the simulation state in either of the simulators, though roll-back may occur inside the CT simulator in order to catch the precise time requested. The CT simulator typically performs a binary search technique to determine zero crossings for event signalling. This process is called microstepping, which is even performed when fixed time step solvers are used.

The overall structure of the co-simulation semantics presented in the remainder of this chapter consists of a central semantic model of the co-simulation engine. It consists of the necessary semantic models for the individual pieces of a co-model. This means that we have three semantic models instantiated for a typical co-model: the co-simulation engine; the Discrete-Event (DE) semantic model of the formalism (VDM) used to describe an embedded digital controller; and the Continuous-Time (CT) semantic model of the formalism (bond graphs) that is used to describe the environment that the embedded controller interacts with. The semantic model of the co-simulation engine is necessarily generic: it needs to be able to support the opaque embedding of other heterogeneous semantic models. In addition, the semantic models of the two simulators are only characterised in this chapter, and we rely on opaque embeddings of them in the semantics of the co-simulation engine.

The semantics of the co-simulation engine can only embed semantic models that conform to certain constraints. Some of these constraints are common to both DE and CT models and these are described in Sect. 13.2.1. The constraints of the CT and DE simulator semantic models are explained in Sects. 13.2.2 and 13.2.3, respectively.

This structure consisting of a central engine coordinating the simulated time and shared state of the overall co-model provides a clear modular division between the generic co-simulation properties and the formalism-specific subject model needs. As a part of this, the actual subject model simulations are delegated to separate semantic models, allowing them to be fit for the needs of those subject models. This, in turn, enables an implementation of the co-simulation engine that allows the use of a variety of different specific simulation engines, as has been done in the Crescendo tool.

13.2.1 Common Semantic Constraints

There are a few common constraints on the semantics of the simulators involved in co-simulation that must be respected. The constraints common to both simulator semantics are as follows:

C1: It must be possible to have the semantics "step" from a given state at a given time to a successor state at some future time, and it must be able to do so in relatively small time increments.

C2: It must calculate the next state of the subject model in a given simulator semantics, taking into account the values from the shared state that have been changed by other simulator semantics.

The first constraint, C1, is required to ensure that it is possible to interleave the simulation of the subject models. The ambiguity that it is able to do so in relatively small increments relates to the subject models (in the Crescendo tool the scale is nanoseconds).

For CT simulator semantics, this effectively means that it must be possible to determine the observable state of a model for any arbitrary (future) point in time, given the granularity defined by the "small time increments" noted above.

Constraint C2 is required to support a shared state between the various subject models in the co-simulation engine. Without this, one may as well just run the individual subject models independently.

13.2.2 Continuous-Time Simulation Semantics

The first type of semantic description that may be used as a simulator semantics in the co-simulation framework is labelled "continuous-time". Semantic descriptions of this type are characterised by a model that is based on a single model state with no hidden or transactional variables. As such, simulators with this sort of semantic model are typically used to model physical systems.

Given the basic assumption that any semantics model in the overall co-simulation framework must make finite steps, there are four primary requirements on a CT semantic model:

CT1: It must be possible to observe the complete state of the semantics for a given point in time.
CT2: It must be possible to set bounds on the maximum duration of each simulation step of a subject model in the semantics.
CT3: It must always produce the actual duration of the simulation step that was taken.
CT4: It must be able to produce "events" as part of a simulation step of a subject model.

The first constraint, CT1, means that there must be no hidden state: the successor state of a CT semantics must be dependent only on the observable state and the time. This lack of pre-determination, in turn, in principle allows changes to shared state variables to have immediate effect on the simulated model.

The second constraint, CT2, is necessary to support the interleaving of CT and DE semantic models. Because it is possible for a DE model to determine that its observable state will change at some given point in the future, we then know that we will need to synchronise the simulator semantic models at that point. Thus, it must be possible to ensure that a step of the semantic model will be no later than that known synchronisation point.

The third constraint, CT3, is required as it may be possible for the step to have taken less time than the maximum allowed by the bound. It may happen that the subject model's successor state in a step is earlier than that bound; this can represent something occurring in that subject model, altering shared state, that requires synchronisation with the other simulation semantics.

Constraint CT4 allows for a mechanism to indicate that something of interest has happened in the simulation of the subject model—an event—without the need to record the event directly in the shared state. This is a sort of side-band signalling mechanism enabling both time-triggered as well as event-triggered behaviour.

13.2.3 Discrete-Event Simulation Semantics

The second type of semantic model for simulation is called the "discrete-event" model. Semantic descriptions of this type are characterised by a model that allows portions of the overall subject model state to be hidden. These hidden portions of the overall model state may be the result of determining the future value of state elements, but not allowing them to be observed until the appropriate point in time. Not surprisingly, this sort of model is typically used to model computational systems, such as embedded controllers.

The requirements on a DE semantic model are concerned with managing the subject model's state and the bounds on the duration of steps of other simulators.

DE1: Every step must indicate the time at which hidden state will next be revealed.

DE2: It must be possible to have the semantics perform a synchronisation step that only exposes hidden state that is ready to be revealed up to a given point in time and updates the internal subject model state with the shared state from the co-simulation engine.

DE3: It must be able to update to a point in time prior to the next point at which hidden state will be revealed.

DE4: It should accept the notification of events that occurred in other simulators.

The first constraint, DE1, on DE semantic models is necessary to indicate to the co-simulation engine the point at which there will next be a change to the shared state due to this subject model. It allows the co-simulation engine to set a bound on the maximum duration of the next step of CT simulation semantics and is necessary to keep the various simulator semantics in the co-simulation engine in step with respect to time.

The second constraint, DE2, anticipates the addition of a fault injection framework and is not strictly necessary for the co-simulation engine itself. In a CT semantic model, as it has no hidden elements, the control and value state of the subject model are always synchronised with the time reported to the co-simulation engine. The DE model, in contrast, may allow hidden elements in its internal state to contain values which are not valid until future points in time. This means that for the subject model to be affected by changes to the shared state from other subject models, it must have a mechanism to update the present internal state before the next internal state is calculated. The initial semantic rules presented in Sect. 13.3.3 do not entirely conform to constraint DE2; however, Sect. 13.4 depends upon the constraint and alters the semantic rules in the required manner.

The third constraint, DE3, allows for this simulation semantics to gracefully handle the other simulation semantics taking steps shorter than expected. The expectation is that a DE model would not normally change any state on such a step. Connected to this is constraint DE4, which requires that the simulation semantics accept the notification of events from the CT model; these events give the subject model an opportunity to react to their presence.

13.3 Co-simulation Semantics

We use the SOS format [77, 78] to present the semantic definitions in this chapter. An SOS description consists of two major elements: a set of type definitions that describe the static structure of the system; and the definitions of the transition relations that describe the behaviour of the system.

The semantics is described using a collection of *transition rules*. Each of these has a number of hypotheses (typically one per line) over a horizontal line. Below the line the conclusion can be reached if all the hypotheses above the line can be reached. It is also possible that such transition rules have side conditions that also need to be satisfied to ensure the validity of applying a specific transition rule. Many of the hypotheses and the conclusion are typically described as transitions which are indicated as something that matches a configuration followed by a line (from left to right with a name above it) and followed again by the new configuration that a system is transformed into.

13.3.1 Structural Operational Semantics

The logical notation used for the assertions in the semantic descriptions is the basic VDM-SL type system and expressions [20, 27].[1] This notation is used to define the static structure of the co-simulation engine and give its behaviour.

In an SOS definition, the static state of a system is modelled as a *configuration* that contains all of the information needed to capture the complete state of a system at any given point. This includes, for the co-simulation semantics we present, information about which simulator semantics was used for the previous step, the

[1]The mathematical syntax will be used for these definitions in order to clearly distinguish it from the use of VDM-RT in the models used in this book.

shared variable state, the current simulated time and the complete internal states of both simulators' semantics (treated opaquely). Configurations are typically given as tuples.

The behaviour of a system is defined through the use of *transition relations*, at least one of which must involve the complete static state configuration. In a fine-grained SOS definition, the overall system behaviour is typically defined using a transition relation from configuration to configuration, though this is not strictly necessary.

The transition relations are defined through the use of inference rule schemata, where each rule's conclusion defines a subset of the entire transition relation. The least relation that satisfies all of the inference rules is taken to be the relation defined.

Consider the following rule:

$$\boxed{\textsf{Example}}\ \frac{\begin{array}{l} P(a,b,a',b') \\ Q(a,b) \\ R(b') \end{array}}{(a,b) \xrightarrow{s} (a',b')}$$

The Example rule here would be a (partial) definition of the $\xrightarrow{s}$ transition relation. The conclusion below the line has four free variables—a, b, a' and b'—into which values may be substituted. For this rule to be true of a given pair of pairs—(a,b) and (a',b') in this case—then the three hypotheses above the line must be satisfied.

In general, the rule applies for any set of values that may be substituted into the free variables so long as all of the hypotheses of the rule are satisfied. This substitution is similar to the pattern matching as done in VDM models and, where a free variable is equated to some other structure, the resolution of the possible values for the free variable is done in a way that allows the constraints in the other structure (concrete values, restrictions to certain types and so on) to hold of the value in the substitution. In practice, however, the rules are often used to determine a successor configuration, that is, the configuration on the right of the transition relation, based on some given predecessor configuration.

For the Example rule, that means that we would start with known values of a and b and then proceed to determine values for a' and b' that satisfy $P(a,b,a',b')$ and $R(b')$. However, we would first check that this rule applies to the given a and b by checking any hypotheses that apply only to a and b; here we check $Q(a,b)$.

The task of finding values for the successor configuration, a' and b' in this case, may be one of simply calculation, or a more complex constraint satisfaction problem. It may also involve choosing between equally valid alternatives for some values—these cases can introduce non-determinism into the semantic model,[2]

[2]Or under-definedness, rather than non-determinism, depending on the perspective and interpretation of the semantic model.

which can be useful but must be resolved during implementation. Finally, it may not be possible to find values for a successor configuration: this may indicate that the predecessor configuration is not a part of the system modelled; that the wrong rule is being applied; that there's an error or missing rule in the semantic model; or that the semantic model has deliberately chosen that the predecessor configuration has no successor.

13.3.2 Co-simulation Static State

Given a co-simulation system that has a CT-type simulator and a DE-type simulator, the overall co-simulation system configuration is given as

$$Config = DE \times CT \times \Sigma \times Time \times Time \times Event\text{-}\mathbf{set} \times Tag \tag{13.1}$$

where

- *DE* is the type of representations of the DE-type simulator, covering all of the possible states that it may reach;
- *CT* is the type of representations of the CT-type simulator, also covering all of the states that it may reach;
- Σ is the representation of the variables shared between the two simulator semantic models and is defined below;
- the first *Time* component of the tuple represents the current simulated time of the overall co-simulation system;
- the second *Time* component represents a time bound that must be respected by CT simulator semantics;
- $Event$-**set** is the set of *Event*s generated by CT semantic models for the DE models to react to; and
- *Tag* is a token that records which of the two semantic models took the last step.

The definitions of the *DE* and *CT* representations are left undefined here; it suffices that they conform to the constraints given in Sect. 13.2. These representations are the overall configurations from their respective semantic models.

The type of the shared variables is a map from an identifier (represented by the inexhaustible set Id_v) to a pair of a value and a tag indicating which simulator semantics controls the value of that variable.

$$\Sigma = Id_v \xrightarrow{m} (Value \times Tag) \tag{13.2}$$

Note that a shared variable ownership is immutable over the course of a co-simulation run. Changing the ownership of a shared variable would imply some significant change to the structure of the subject models in the co-simulation.

The tag is simply a structure-less token, either ⟨DE⟩ or ⟨CT⟩.

$$\boxed{\textsf{DE Step}}\ \frac{\begin{array}{l}(de,\sigma,\tau,events) \xrightarrow{de} (de',\sigma',\tau_b') \\ \sigma'' = mergeStates(\sigma,\sigma',\langle \mathrm{DE}\rangle)\end{array}}{(de,ct,\sigma,\tau,\tau_b,events,\langle \mathrm{CT}\rangle) \xrightarrow{cs} (de',ct,\sigma'',\tau,\tau_b',events,\langle \mathrm{DE}\rangle)}$$

$$\boxed{\textsf{CT Step}}\ \frac{\begin{array}{l}(ct,\sigma,\tau_b) \xrightarrow{ct} (ct',\sigma',\tau',events') \\ \sigma'' = mergeStates(\sigma,\sigma',\langle \mathrm{CT}\rangle)\end{array}}{(de,ct,\sigma,\tau,\tau_b,events,\langle \mathrm{DE}\rangle) \xrightarrow{cs} (de,ct',\sigma'',\tau',\tau_b,events',\langle \mathrm{CT}\rangle)}$$

Fig. 13.2 Inference rules for the behaviour of the co-simulation engine

$$Tag = \langle \mathrm{DE}\rangle \mid \langle \mathrm{CT}\rangle \tag{13.3}$$

13.3.3 Co-simulation Behaviour

With the static state defined, we can now define the transition relation for the co-simulation semantics, $\xrightarrow{cs}$.

$$\xrightarrow{cs}:\ Config \times Config \tag{13.4}$$

Note that the colon, above, is a type assertion; $\xrightarrow{cs}$ is a relation over the type of configurations, *Config*.

We give two inference rules to define the high-level behaviour of the co-simulation semantics in Fig. 13.2. The first rule, DE Step, describes the behaviour that is dependent on the semantics of the discrete-event simulator; the second rule, CT Step, handles the corresponding case when the behaviour is dependent on the semantics of the continuous-time simulator.

Starting with the DE Step rule, consider first the *Config* tuple on the left of the $\xrightarrow{cs}$ relation. This tuple is given in terms of its components to allow us to name the components for use in the rule.

Moving our focus to the first hypothesis of the rule, we see some of the components of the *Config* tuple being used in a new transition relation, $\xrightarrow{de}$. This transition relation is defined as

$$\xrightarrow{de}:\ (DE \times \Sigma \times Time \times Event\text{-}\mathbf{set}) \times (DE \times \Sigma \times Time) \tag{13.5}$$

and it represents a single step in the discrete-event semantics. Informally, the left side of the $\xrightarrow{de}$ transition is a tuple of a DE simulator state, shared variable state, a new simulated time for the DE to update to and a set of events that happened between the last point at which the DE semantics took a step and the new simulated time; the right side is a tuple of the new DE simulator state, the new values of shared variables and a new time bound.

$$\begin{aligned}&\mathit{mergeStates}\colon \Sigma \times \Sigma \times \mathit{Tag} \to \Sigma\\&\mathit{mergeStates}(\sigma_o, \sigma_t, \mathit{tag}) == \sigma_o \dagger \{\mathit{id} \mapsto \sigma_t(\mathit{id}) \mid \mathit{id} \in \mathbf{dom}\, \sigma_t \wedge \sigma_o(\mathit{id}).\#2 = \mathit{tag}\}\end{aligned}$$

Fig. 13.3 Function to merge two shared states, taking only the values paired with a specific tag value

The second hypothesis merges the new values in the shared variable mapping what were produced by the DE semantics into the overall co-simulation shared state. The details of the merge are encapsulated into the *mergeStates* function, defined in Fig. 13.3. This hypothesis is separate from the DE semantics to emphasise that this synchronisation is, properly, the responsibility of the overall co-simulation semantics and not the individual simulator semantics. One important property that is maintained by this separation is that each simulation semantics may only affect the values that it "owns", as marked by the control tag in the state.

So, the DE Step rule defines the behaviour of the co-simulation semantics when executing a DE step directly in terms of the behaviour of the DE semantics, with a bit of extra mechanism to ensure that the overall shared state is updated.

The second rule in Fig. 13.2, CT Step, is structured in a similar manner to the first, but uses the $\xrightarrow{ct}$ transition relation to model the behaviour of the CT semantics as it performs a single step. This transition relation is defined as

$$\xrightarrow{ct}\colon\ (CT \times \Sigma \times \mathit{Time}) \times (CT \times \Sigma \times \mathit{Time} \times \mathit{Event}\text{-}\mathbf{set}) \tag{13.6}$$

where, informally, the left side of the transition relation is the CT simulator state, the shared variable state and a time bound; and the right side of the transition relation is the new CT simulator state, the modified shared variables, the time up to which the CT semantics actually reached and a (possibly empty) set of events generated during that step.

The primary advantage of this semantic description of co-simulation is that it isolates the semantics of the two simulators as much as possible, allowing the semantics for each simulator to be defined independently.

13.3.4 Simulator Properties and Their Transition Relations

For the common constraints, C1 and C2, it is clear that both the $\xrightarrow{de}$ and $\xrightarrow{ct}$ transition relations are satisfactory. The ability of the semantic models to step from state to state is inherent in SOS descriptions, thus satisfying C1. Satisfaction of C2 is actually handled internally within the individual definitions of $\xrightarrow{de}$ and $\xrightarrow{ct}$ that we must be satisfied with the simple structural compliance present. Specifically, the shared co-simulation state is in the signature of the transition relation, allowing a definition of the transition to use and update it.

Focusing on $\xrightarrow{ct}$ transition, it must satisfy the four CT constraints, CT1–CT4. Two, CT1 and CT3, are dependent on the internal definition of the $\xrightarrow{ct}$ transition, and the external structure cannot speak to this. However, CT2 requires that the internal definition of $\xrightarrow{ct}$ be aware of a maximum duration and this appears structurally in the left-hand configuration. Finally, CT4 requires that the semantic description be allowed to produce events, and this appears directly in the structure of the right-hand configuration.

For the $\xrightarrow{de}$ transition, the four DE constraints, DE1–DE4, must be satisfied. Constraints DE1, DE3 and DE4 are satisfied as far as possible by the structure of the signature of $\xrightarrow{de}$; what remains is the responsibility of the internal definition of $\xrightarrow{de}$ to handle. For constraint DE2, the signature of $\xrightarrow{de}$ can support this, as it is possible to indicate that a synchronisation-only step is required by defining an event with such a special meaning. Beyond the structure, again, the rest is handled by the internal definition of $\xrightarrow{de}$.

Possible internal definitions for the $\xrightarrow{de}$ transition may be found in [61], and a description for the $\xrightarrow{ct}$ may be found in [24]. In this book, the $\xrightarrow{de}$ transition corresponds to the semantics of the executable subset of VDM-RT while the $\xrightarrow{ct}$ transition corresponds to the simulation of the bond graphs.

13.4 Adding Fault Injection Semantics to the Co-simulation

The co-simulation engine used in the Crescendo tool presents an opportunity to create a fault injection framework that is independent of any of the constituent simulators. This framework only has access to variables that have been explicitly shared between the simulation engines, but this is sufficient for the creation of fault injection and external event scripting.

The CSL is introduced in the Crescendo user documentation and is a minimal language intended to allow for changes to the shared state of the simulators when certain conditions are met. The basic structure of a script—a series of commands in CSL—is just a sequence of triggers, each with a firing condition, body and optional reversions.

The CSL is essentially a reactive language, and a script is a sequence of triggers. A trigger consists of a test expression and a set of assignments to the shared state and may also optionally have a list of variables to revert to their values prior to the trigger. A trigger may be set to react every time its test expression is true, or it may react only once during the entire co-simulation run.

In the basic co-simulation cycle, the DE simulator runs first, followed by the CT simulator, and execution continues, alternating between the two simulators. To add CSL into this cycle, we need to split the DE step and add a new CSL step. We must also extend the configuration, *Config*, with a representation of the CSL script, and the *Tag* type of tokens with $\langle \text{CSL} \rangle$ and $\langle \textsc{Sync} \rangle$ to include tags for the CSL.

$$Config = DE \times CT \times CSL \times \Sigma \times Time \times Time \times Event\text{-}\mathbf{set} \times Tag$$

$$\boxed{\textsf{DE Step}}\ \frac{\begin{array}{l}(de, \sigma_o, \tau, events) \xrightarrow{de} (de', \sigma_s, \tau_b') \\ \sigma_o' = mergeStates(\sigma_o, \sigma_s, \langle \mathrm{DE} \rangle) \\ \tau_b'' = min(\tau_b, \tau_b')\end{array}}{(de, ct, csl, \sigma_o, \tau, \tau_b, events, \langle \mathrm{CSL} \rangle) \xrightarrow{cs} (de', ct, csl, \sigma_o', \tau, \tau_b'', events, \langle \mathrm{DE} \rangle)}$$

$$\boxed{\textsf{DE Sync}}\ \frac{\begin{array}{l}(de, \sigma_o, \tau, \{\langle \mathrm{S{\scriptsize YNC}} \rangle\}) \xrightarrow{de} (de', \sigma_s, \tau_b') \\ \sigma_o' = mergeStates(\sigma_o, \sigma_s, \langle \mathrm{DE} \rangle) \\ \tau_b'' = min(\tau_b, \tau_b')\end{array}}{(de, ct, csl, \sigma_o, \tau, \tau_b, events, \langle \mathrm{CT} \rangle) \xrightarrow{cs} (de', ct, csl, \sigma_o', \tau, \tau_b'', events, \langle \mathrm{S{\scriptsize YNC}} \rangle)}$$

$$\boxed{\textsf{CT Step}}\ \frac{\begin{array}{l}(ct, \sigma_o, \tau_b) \xrightarrow{ct} (ct', \sigma_s, \tau', events') \\ \sigma_o' = mergeStates(\sigma_o, \sigma_s, \langle \mathrm{CT} \rangle)\end{array}}{(de, ct, csl, \sigma_o, \tau, \tau_b, events, \langle \mathrm{DE} \rangle) \xrightarrow{cs} (de, ct', csl, \sigma_o', \tau', \tau_b, events', \langle \mathrm{CT} \rangle)}$$

$$\boxed{\textsf{CSL Step}}\ \frac{\begin{array}{l}(csl, \sigma_o, \tau) \xrightarrow{csl} (csl', \sigma_o', \tau_b') \\ \tau_b'' = min(\tau_b, \tau_b')\end{array}}{(de, ct, csl, \sigma_o, \tau, \tau_b, events, \langle \mathrm{S{\scriptsize YNC}} \rangle) \xrightarrow{cs} (de, ct, csl', \sigma_o', \tau, \tau_b'', events, \langle \mathrm{CSL} \rangle)}$$

Fig. 13.4 New semantic rules for the behaviour of the simulators in the co-simulation semantics

When splitting the DE Step rule from Fig. 13.2, we replace it with the version in Fig. 13.4. The difference between the two rules is that the extended version requires that the *Tag* in the left-hand configuration be ⟨CSL⟩ instead of ⟨CT⟩ and that the new time bound, τ_b'', must be the least between that generated by the DE semantics, in τ_b', and last execution of the CSL script, in τ_b.

We also add a new rule, DE Sync, that does a $\xrightarrow{de}$ transition with a SYNC event marked; the intent is to signal to the internal definition of the $\xrightarrow{de}$ transition that it should perform synchronisation of hidden state, and not perform any new computation, as per Constraint DE2. This rule also takes the least available time bound for τ_b''.

For completeness, Fig. 13.4 also shows the updated CT Step rule and definition of the updated *Config*.

The transition, $\xrightarrow{csl}$, for the CSL can be defined as

$$\xrightarrow{csl}:\ (CSL \times \Sigma \times Time) \times (CSL \times \Sigma \times Time) \tag{13.7}$$

where *CSL* is the type representing CSL scripts. The two *Time* values correspond, respectively, to the simulated time in the subject model at which the CSL script is evaluated and a time bound indicating the next simulated time at which the CSL may be triggered.

The necessary semantic rule to use the $\xrightarrow{csl}$ transition for the CSL is the last rule given in Fig. 13.4. This rule follows the pattern of the rest of the rules in Fig. 13.4, except that it does not need to merge states as it operates on the shared state directly. Further, given that the CSL is intended to allow changes to the shared state of both simulators, there is no need to handle ownership of variables.

Taking the rules in Fig. 13.4 as a whole, we can see that the round-robin cycle is preserved; in order, the rules proceed from the DE Step rule to CT Step to DE Sync to CSL Step and then back around.

13.5 Semantics of the CSL

The intended execution of a CSL script is straightforward: for a setpoint in time, each trigger is sequentially processed, modifying the shared state as necessary. The semantic model given below reflects this as the transition relation recursively processes the sequence of triggers, producing a new shared state when given an empty sequence.

13.5.1 Top-Level CSL Structures

The active structure of the CSL in the co-simulation semantics is the *CSL* construct.

$$\begin{array}{lrcl} CSL :: & ts & : & Script \\ & ms & : & Marker^* \end{array}$$

The *CSL* construct contains the script of CSL triggers and a sequence of the same length to hold a minimal amount of state for each trigger.

$$Script = Trigger^*$$

Each trigger state—a *Marker*—is either **nil**, a constant DONE or a pair of a time value and a state.

$$Marker = [\textsc{Done} \mid Time \times \Sigma]$$

The initialisation of the *CSL* construct is handled by the $\xrightarrow{cslinit}$ transition,

$$\xrightarrow{cslinit} : Script \times CSL \tag{13.8}$$

with its definition given in Fig. 13.5. The initialisation rule simply transforms a fault injection script into the *CSL* object used by the main CSL rules. The top-level rule is given in Fig. 13.6

The *Trigger* construct is defined as

Fig. 13.5 CSL initialisation rule

$$\boxed{\textsf{CSL Init}}\ \frac{\textbf{elems}\,\mathit{markers} = \{\textbf{nil}\} \qquad \textbf{len}\,\mathit{markers} = \textbf{len}\,\mathit{script}}{\mathit{script} \xrightarrow{\mathit{cslinit}} \mathit{mk\text{-}CSL}(\mathit{script}, \mathit{markers})}$$

Fig. 13.6 The top-level semantic rule for CSL execution

$$\boxed{\textsf{CSL Top}}\ \frac{\begin{array}{l}\sigma = \{id \mapsto value \mid \sigma_o(id) = (tag, value)\} \\ (ts, ms, \tau, \sigma) \xrightarrow{trig} (ms', \sigma') \\ \sigma_o' = \{id \mapsto (\sigma_o(id), value) \mid \sigma'(id) = value\} \\ \tau_b = calculateCSLTimeBound(ts, ms')\end{array}}{(mk\text{-}CSL(ts, ms), \sigma_o, \tau) \xrightarrow{csl} (mk\text{-}CSL(ts, ms'), \sigma_o', \tau_b)}$$

$$\begin{array}{rrcl} Trigger :: & once & : & \mathbb{B} \\ & test & : & Expr \\ & dur & : & [Time] \\ & body & : & Stmt^* \\ & rev & : & Id_v\textbf{-set} \end{array}$$

A *Trigger* has a flag, *once*, indicating whether or not it may occur only once or many times; a condition, *test*, which is evaluated to determine if the trigger should fire; an optional duration field, *dur*, which if present requires that the condition hold over the given duration; a body field, *body*, that holds assignment and print statements to be executed when the trigger fires; and a field, *rev*, that stores a set of variable names to be reverted to their values prior to the trigger firing.

The basic execution of a CSL script is modelled by the $\xrightarrow{csl}$ relation,

$$\xrightarrow{csl}: (CSL \times \Sigma_o \times Time) \times (CSL \times \Sigma_o \times Time)$$

and that rule delegates most of the actual work of executing the triggers to the $\xrightarrow{trig}$ transition relation. Of the rest of the rule, the first and third hypotheses deal with the flattening and rebuilding of the shared state, and the final hypothesis calculates a time bound for the co-simulation based on the triggers that are actively testing a condition over a duration.

The $\xrightarrow{trig}$ transition relation,

$$\xrightarrow{trig}: (Trigger^* \times Marker^* \times Time \times \Sigma) \times (Marker^* \times \Sigma)$$

only alters the sequence of *Marker*s and the shared state; the actual triggers are never altered. The three basic rules for a trigger with a simple test (no duration) are shown in Figs. 13.7, 13.8 and 13.9.

Reading the Exec rule in Fig. 13.7, we see that the configuration is structured to pattern match the heads of the sequences of triggers and markers, ensuring that we only execute a trigger that is not already active. Then, the first hypothesis pattern matches the first trigger on the sequence to ensure that it is one without a duration. The second hypothesis ensures that the trigger's condition has been met; evaluation of the expression *test* is done using the semantic evaluation function $[\![\cdot]\!]$, and the result is, in turn, given a τ and a σ to evaluate the expression in the time and state context down to its value. The third hypothesis delegates the execution of the

Fig. 13.7 The Exec semantic rule for executing triggers

$$\boxed{\mathsf{Exec}}\ \frac{\begin{array}{l} trigger = \textit{mk-Trigger}(once, test, \mathbf{nil}, body, rev) \\ [\![test]\!]\tau\sigma = \mathbf{true} \\ (body, \sigma) \xrightarrow{tstmt} \sigma' \\ (ts, ms, \tau, \sigma') \xrightarrow{trig} (ms', \sigma'') \end{array}}{([trigger] \frown ts, [\mathbf{nil}] \frown ms, \tau, \sigma) \xrightarrow{trig} ([(\tau, \sigma)] \frown ms', \sigma'')}$$

Fig. 13.8 The Skip semantic rule for executing triggers

$$\boxed{\mathsf{Skip}}\ \frac{\begin{array}{l} trigger = \textit{mk-Trigger}(once, test, dur, body, rev) \\ [\![test]\!]\tau\sigma = \mathbf{true} \\ dur \in Time \;\Rightarrow\; \tau\text{-}\tau_0 > dur \\ (ts, ms, \tau, \sigma') \xrightarrow{trig} (ms', \sigma'') \end{array}}{([trigger] \frown ts, [(\tau_0, \sigma_0)] \frown ms, \tau, \sigma) \xrightarrow{trig} ([(\tau_0, \sigma_0)] \frown ms', \sigma'')}$$

Fig. 13.9 The After semantic rule for executing triggers

$$\boxed{\mathsf{After}}\ \frac{\begin{array}{l} trigger = \textit{mk-Trigger}(\mathbf{false}, test, dur, body, rev) \\ [\![test]\!]\tau\sigma = \mathbf{false} \\ dur \in Time \;\Rightarrow\; \tau\text{-}\tau_0 > dur \\ \sigma' = \sigma \dagger (rev \lhd \sigma_0) \\ (ts, ms, \tau, \sigma') \xrightarrow{trig} (ms', \sigma'') \end{array}}{([trigger] \frown ts, [(\tau_0, \sigma_0)] \frown ms, \tau, \sigma) \xrightarrow{trig} ([\mathbf{nil}] \frown ms', \sigma'')}$$

body of the trigger to the $\xrightarrow{tstmt}$ transition relation, producing a potentially altered shared state. The last hypothesis executes the rest of the sequence of triggers.[3] The conclusion of this rule places a pair containing the present time—that is, when the trigger fired—and the initial state at the head of the sequence of markers in the resulting configuration. This indicates to future evaluations of this trigger that the trigger has fired and is waiting for the condition to become false.

The Skip rule in Fig. 13.8 just skips over triggers that have a pair as their marker and a true condition; for triggers with durations, the rule only applies when the duration has been exceeded (third hypothesis). When the marker is a pair, this indicates that the trigger has been active; furthermore, that the condition is still true means that it is not yet time to revert any variables.

The After rule in Fig. 13.9 handles the case where the trigger's condition has become false and there are shared variables to revert. This rule applies to all repeating triggers, regardless of whether or not they have a duration. The first two hypotheses pattern match the trigger at the head of the sequence and that the condition is false. The third hypothesis is a guard for triggers with a duration, to ensure that we do not revert shared variables when we should instead be cancelling a trigger with a duration that was still waiting for the necessary time to pass. The fourth hypothesis does the necessary reversion of the state, pulling the values of the variables to be reverted from the state at the start of the trigger, σ_0; note that if the trigger did not have any reverts specified, then the set *rev* will be empty and

[3]The base case—an empty sequence of triggers—is handled by the rule Base in Fig. 13.10.

$$\boxed{\text{Base}}\ \frac{}{([\,], ms, \tau, \sigma) \xrightarrow{trig} (ms, \sigma)}$$

$$\boxed{\text{Done}}\ \frac{(ts, ms, \tau, \sigma) \xrightarrow{trig} (ms', \sigma')}{([trigger] \curvearrowright ts, [\text{DONE}] \curvearrowright ms, \tau, \sigma) \xrightarrow{trig} ([\text{DONE}] \curvearrowright ms', \sigma')}$$

$$\boxed{\text{Once after}}\ \frac{\begin{array}{l} trigger = mk\text{-}Trigger(\mathbf{true}, test, dur, body, rev) \\ [\![test]\!]\tau\sigma = \mathbf{false} \\ dur \in Time \;\Rightarrow\; \tau\text{-}\tau_0 > dur \\ \sigma' = \sigma \dagger (rev \lhd \sigma_0) \\ (ts, ms, \tau, \sigma') \xrightarrow{trig} (ms', \sigma'') \end{array}}{([trigger] \curvearrowright ts, [(\tau_0, \sigma_0)] \curvearrowright ms, \tau, \sigma) \xrightarrow{trig} ([\text{DONE}] \curvearrowright ms', \sigma'')}$$

Fig. 13.10 The remaining semantic rules for executing triggers

$\sigma' = \sigma$. The fifth hypothesis is the usual recursive step of executing the rest of the triggers.

Those are the three basic rules for executing triggers in the CSL semantics. There are ten semantic rules in total and they have been constructed so that there is no situation where multiple rules apply.

Of the remaining semantic rules, some of the base cases are given in Fig. 13.10. The first, Base, is only necessary to handle the base case of processing a sequence: that is, when the sequence is empty. Two rules, Done and Once after, deal with the non-repeating triggers: the former skips non-repeating triggers that have already triggered; and the latter places a DONE constant in the marker after the trigger completes.

The last four remaining rules in Fig. 13.11, Duration init, Duration pending, Duration cancel and Duration exec, handle the specific cases where a trigger with a duration on its conditional must, respectively, start monitoring the duration in which the conditional has been true; do nothing between the condition becoming true but before the duration has passed; stop monitoring the duration if the condition goes false before the duration passes; and to execute the trigger body when the duration is met.

13.5.2 CSL Statement and Expression Semantics

The rules for statements and expressions are included only to give a complete semantic model of the CSL. There are only three statements—assignment, output and quit—and the expression evaluation is essentially a subset VDM's expressions (with the singular addition of a special identifier to obtain the current time value).

$$\boxed{\text{Duration init}}\ \frac{\begin{array}{l} trigger = mk\text{-}Trigger(once, test, dur, body, rev) \\ [\![test]\!]\tau\sigma = \mathbf{true} \\ dur \in Time \\ (ts, ms, \tau, \sigma) \xrightarrow{trig} (ms', \sigma') \end{array}}{([trigger] \curvearrowright ts, [\mathbf{nil}] \curvearrowright ms, \tau, \sigma) \xrightarrow{trig} ([(\tau, \sigma)] \curvearrowright ms', \sigma')}$$

$$\boxed{\text{Duration cancel}}\ \frac{\begin{array}{l} trigger = mk\text{-}Trigger(once, test, dur, body, rev) \\ [\![test]\!]\tau\sigma = \mathbf{false} \\ \tau\text{-}\tau_0 \leq dur \\ (ts, ms, \tau, \sigma) \xrightarrow{trig} (ms', \sigma') \end{array}}{([trigger] \curvearrowright ts, [(\tau_0, \sigma_0)] \curvearrowright ms, \tau, \sigma) \xrightarrow{trig} ([\mathbf{nil}] \curvearrowright ms', \sigma')}$$

$$\boxed{\text{Duration pending}}\ \frac{\begin{array}{l} trigger = mk\text{-}Trigger(once, test, dur, body, rev) \\ [\![test]\!]\tau\sigma = \mathbf{true} \\ \tau\text{-}\tau_0 < dur \\ (ts, ms, \tau, \sigma) \xrightarrow{trig} (ms', \sigma') \end{array}}{([trigger] \curvearrowright ts, [(\tau_0, \sigma_0)] \curvearrowright ms, \tau, \sigma) \xrightarrow{trig} ([(\tau_0, \sigma_0)] \curvearrowright ms', \sigma')}$$

$$\boxed{\text{Duration exec}}\ \frac{\begin{array}{l} trigger = mk\text{-}Trigger(once, test, dur, body, rev) \\ [\![test]\!]\tau\sigma = \mathbf{true} \\ \tau\text{-}\tau_0 = dur \\ (body, \tau, \sigma) \xrightarrow{tstmt} \sigma' \\ (ts, ms, \tau, \sigma') \xrightarrow{trig} (ms', \sigma'') \end{array}}{([trigger] \curvearrowright ts, [(\tau_0, \sigma_0)] \curvearrowright ms, \tau, \sigma) \xrightarrow{trig} ([(\tau_0, \sigma_0)] \curvearrowright ms', \sigma'')}$$

Fig. 13.11 Duration rules for the CSL

13.5.2.1 Structure

Statements in CSL are represented by the *Stmt* type and consist of assignment statements, output statements to print message to the log and the quit statement.

$$Stmt = Assign \mid Output \mid \textsc{Quit}$$

Assignment statements have the state variable that will be modified and an expression that is evaluated to a value assigned.

$$\begin{array}{lrcl} Assign :: & id & : & Id_v \\ & e & : & Expr \end{array}$$

Output statements may print a message to one of three logs: the regular message log, PRINT, the warnings log, WARN, or the error log, ERROR.

$$\begin{array}{lrcl} Output :: & target & : & \textsc{Print} \mid \textsc{Warn} \mid \textsc{Error} \\ & message & : & String \end{array}$$

Expressions in CSL are represented by the *Expr* type and consist of the special variable TIME, time values, *Time*, Boolean values, real number values, identifiers, unary expressions and binary expressions.

$$Expr = \textsc{Time} \mid Time \mid \mathbb{B} \mid \mathbb{R} \mid Id_v \mid UnaryExpr \mid BinaryExpr$$

Unary expressions in CSL include the arithmetic operators for negation and absolute value, the rounding functions floor and ceiling and the boolean negation operator.

$$\begin{array}{lrcl} UnaryExpr :: & op & : & \text{-} \mid \textsc{abs} \mid \textsc{floor} \mid \textsc{ceil} \mid \textsc{not} \\ & e & : & Expr \end{array}$$

Binary expressions in CSL include the usual logical operators for conjunction, disjunction, implication and bi-implication; the basic arithmetic comparison operators; the usual arithmetic addition, subtraction, multiplication and division; and the integer modulo and division operators.

$$\begin{array}{lrcl} BinaryExpr :: & op & : & \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \mid < \mid \leq \mid \geq \mid > \mid = \mid \neq \mid + \mid \text{-} \mid \times \mid \div \mid \textsc{mod} \mid \textsc{div} \\ & a & : & Expr \\ & b & : & Expr \end{array}$$

Finally, the transition relation for statement execution in CSL is defined as $\xrightarrow{tstmt}$, which relates a tuple containing a sequence of statements, a current time value and an initial state, Σ, to a final state.

$$\xrightarrow{tstmt}: (Stmt^* \times Time \times \Sigma) \times \Sigma$$

13.5.2.2 Statement Rules

There are four rules to define $\xrightarrow{tstmt}$, giving the behaviour of statements in CSL.

$$\boxed{\text{Stmt base}}\ \frac{}{([\,], \tau, \sigma) \xrightarrow{tstmt} \sigma}$$

The first rule defines the base case behaviour for sequences of statements, just giving the state on the left of the transition relation as the final state.

$$\boxed{\text{Stmt Assign}}\ \frac{\begin{array}{l} \sigma' = \sigma \dagger \{id \mapsto [\![e]\!]\tau\sigma\} \\ (rest, \sigma') \xrightarrow{tstmt} \sigma'' \end{array}}{([mk\text{-}Assign(id, e)] \curvearrowright rest, \tau, \sigma) \xrightarrow{tstmt} \sigma''}$$

The rule for assignment statements evaluates the expression given the current time and state, and then executes the remaining statements in an appropriately modified state.

$$\boxed{\text{Stmt Print}}\ \frac{\begin{array}{l} output(target, message) \\ (rest, \sigma) \xrightarrow{tstmt} \sigma \end{array}}{([mk\text{-}Print(target, message)] \curvearrowright rest, \tau, \sigma) \xrightarrow{tstmt} \sigma}$$

The function *output*() in the Stmt Print rule is left undefined; the intent is that it maps to some implementation-dependent logging facility.

The third type of statement, QUIT, does not have a semantic rule defined here. The quit statement is implementation-dependent as it is intended to stop simulation of the entire co-model.

13.5.2.3 Expression Evaluation

Expression evaluation in the CSL is defined in the usual way, though we include access to the current time value using the special identifier TIME. Note, also, that the set of time values, *Time*, is a subset of the real numbers.

The semantics of expression evaluation is given as a function, $[\![\cdot]\!]$, over three parameters that results in either a Boolean value or a real number. The first parameter is the expression to be evaluated, the second is the time at which the expression is evaluated and the third is the state in which it is evaluated.

$[\![\cdot]\!] : Expr \times Time \times \Sigma \to (\mathbb{B} \mid \mathbb{R})$

The specific cases in the definition of $[\![\cdot]\!]$ are given below.

$$\begin{aligned}
[\![\mathbf{true}]\!]\tau\sigma &= \mathbf{true} \\
[\![\mathbf{false}]\!]\tau\sigma &= \mathbf{false} \\
[\![\text{TIME}]\!]\tau\sigma &= \tau \\
[\![r]\!]\tau\sigma &= r && \Leftrightarrow\ r \in \mathbb{R} \\
[\![id]\!]\tau\sigma &= \sigma(id) && \Leftrightarrow\ id \in \mathbf{dom}\,\sigma \\
[\![mk\text{-}UnaryExpr(op, e)]\!]\tau\sigma &= op([\![e]\!]\tau\sigma) \\
[\![mk\text{-}BinaryExpr(op, e_1, e_2)]\!]\tau\sigma &= ([\![e_1]\!]\tau\sigma)\ op\ ([\![e_2]\!]\tau\sigma)
\end{aligned}$$

The first four cases deal with constant values: the Boolean and real values evaluate to themselves (the first, second and fourth lines), and the special constant TIME evaluates the given time value, τ. The fifth case deals with variable identifiers by returning their corresponding value from the state. The last two cases deal with operators by applying the operation to the result(s) of evaluating their operand(s).

13.6 Conclusion

In order to be able to trust the outcomes of model-based analyses, we need languages and tools that have a formal semantics, and the semantics must form the basis of the tooling. For co-simulation, the semantic basis is perhaps more intricate than that of monodisciplinary modelling because of the need to be precise about the management of communication and the time in the collaborating simulations. The overall manner in which the Crescendo co-simulation works was presented at the start of this chapter, and that was used as a starting point to give the general semantic constraints under which a co-simulation framework must operate. With those constraints in mind, we described a modular operational semantic model that gives the behaviour of co-simulation while leaving the behaviour of the individual simulators abstract. We then presented an extension to the co-simulation semantic model that includes support for fault injection. This was presented as an extension to the whole co-simulation semantics to allow the fault injection to be abstract with respect to the simulators. We then gave the operational semantics of CSL, as that was the fault injection language implemented for the Crescendo tool. Readers interested in a detailed presentation of the VDM-RT simulator are invited to review the execution semantics [24].