Hint-Based Configuration of Co-simulations with Algebraic Loops

Bentley James Oakes[1,2], Cláudio Gomes[1,2], Franz Rudolf Holzinger[3], Martin Benedikt[3], Joachim Denil[1,2], and Hans Vangheluwe[1,2]

[1] University of Antwerp, Belgium
[2] Flanders Make vzw, Belgium
[3] VIRTUAL VEHICLE Research GmbH, Austria

Abstract. Co-simulation is a powerful technique for performing full-system simulation. Multiple black-box models and their simulators are combined together to provide the behaviour for a full system. However, the black-box nature of co-simulation and potentially infinite configuration space means that configuration of co-simulations is a challenging problem for today's practitioners.
Our previous work on co-simulation configuration operated on the notion of hints, which allow system engineers to encode their knowledge and insights about the system. These hints, combined with state-of-the-art best practices, can then be used to semi-automatically configure the co-simulation.
We summarize our previous hint-based configuration work here, and explore the challenging problem of scheduling co-simulations which contain algebraic loops. Solving or "breaking" these loops is required for scheduling, yet this breaking process can induce errors in the co-simulation. This work formalizes this scheduling problem, presents our insights gained about the problem, and details an optimal search algorithm as well as greedy scheduling algorithms. These heuristic algorithms are evaluated on (synthetic) co-simulation scenarios to determine their relative speedup and optimality.

1 Introduction

Cyber-Physical Systems (CPS) marry the complexities of software with the realities of the physical world [24], and are becoming essential systems in today's world. For example, an airplane or self-driving car relies on safety-critical communication between sensors, controller software, and actuators. A large driver in the complexity of CPS is that their analysis spans multiple domains. Simulating these therefore requires integrating heterogeneous models. For example, the integration of multi-body system models with communication network models.
The technique of *co-simulation* is designed to combine multiple *co-simulation units* (simulators, each with their own model), in order to compute the behavior of the combined models over time [16, 23]. Each unit has its own interface for getting/setting inputs and outputs, and for computing the behaviour of its model over a given interval of time. An example of such an interface is the Functional Mockup Interface (FMI) Standard [7, 10]. A *master algorithm* is then responsible for scheduling the execution and communication of each co-simulation unit.
Co-simulation is therefore very promising for representing: a) systems assembled from models in various domains, each with their own most appropriate simulator,

and b) black-box models which hide internal details, allowing third-party units to be integrated together, as happens in supplier-integrator relationships in industry. These key benefits have led to multiple modeling and simulation tools allowing the import and export of units implementing the FMI Standard [9], and a wide variety of applications [16].
However, it can be difficult to ensure that the results produced by a co-simulation can be trusted. This is due to not only the black box nature of units, but also to the many ways in which a co-simulation can be computed (i.e., communication frequency between differential-equation-based units, event propagation order, etc. . .). The correct configuration of a co-simulation also depends on the numerical properties of the participating units. This challenge is aggravated when units themselves may be modified as part of an optimization loop (e.g., design space exploration) and/or impact analysis of sub-model refinements, because the co-simulation user may be unaware of how to account for these modifications in the master algorithm.

Prior Work. Our earlier work [18] focused on the core challenge that *users do not always know how to configure the co-simulation* [25]. To tackle this, we proposed a method, and a tool called HintCO which is summarized in Section 2 and available online [13]. In this tool, a user or engineer can write "hints" about the co-simulation and involved units. HintCO then applies state of the art heuristics to configure and run multiple promising co-simulations. This is similar to design-space exploration techniques [28, 30]. This approach works well in practice because users usually can tell what properties a correctly configured co-simulation should satisfy. This is demonstrated by an industrial case study in [18], where state of the art co-simulation algorithms failed to produce expected 'smooth' (non-oscillatory) results. After specifying a few hints, the top candidate results produced by HintCO were smooth.

Motivation. One limitation of HintCO was the assumption that the co-simulation unit couplings do not form *algebraic loops*, as explained in Section 3. However, when differential-algebraic-equation-based units are coupled, algebraic loops can be formed [1]. The ideal technique to solve algebraic loops in co-simulations is fixed point iteration (see, e.g., [15, Section 4] and [26]). However, this technique requires that units support state rollback, which is an optional feature of the FMI standard and is therefore seldom implemented currently. Therefore, the most common technique is to "break" the algebraic loop by employing extrapolations in one (or more) variables in the loop [3, 5]. Naturally, variables have different dynamics, hence, care must be taken when choosing the variables to break the loop [19].
Our prior work [18] naïvely generates all possible ways in which algebraic loops can be broken, without regard for the dynamics of the variables involved. In the current work, we build on [19] to formalize the problem of breaking algebraic loops in co-simulations, and propose an optimal algorithm to solve it, plus a few cost-effective approximation algorithms.

Contributions. Our contributions in this paper are: a) a formalization of the problem of breaking algebraic loops in co-simulations, b) an optimal, but costly, algorithm to solve it, and c) multiple cost-effective heuristic algorithms.

1.1 Paper Layout

The next section (Section 2) will provide a brief introduction to the HintCO framework, including the hints and how they shape the search space for finding a correct co-simulation master algorithm. While HintCO has been shown to be effective for an industrial case [18], in Section 3 we demonstrate an example with an algebraic loop, where the previous version of the HintCO framework was unable to efficiently schedule this co-simulation.

Section 4 formalizes the essential components of our approach. We introduce co-simulation and its involved concepts, as well as the background for our improved approaches to co-simulation scheduling. The concrete problem of scheduling co-simulations in the presence of algebraic loops is formalized in Section 5 and a optimal yet costly algorithm is provided.

Candidate greedy algorithms for scheduling co-simulations with algebraic loops are presented in Section 6, and evaluated on synthetic examples in Section 7. Following this, Section 8 will discuss related work in the field and compare our approach to past approaches. Finally, we conclude in Section 9 with a summary of our research and the steps to extend our framework further.

2 HintCO Framework

This section briefly introduces relevant aspects of the HintCO framework, such that the importance of the contributions made in this paper to the automated configuration of co-simulations can be appreciated. In particular, this section adapts text from [18] to briefly describe the problem statement tackled by the HintCO framework and the elements of the HintCO workflow. This includes exemplifying some of hints the user can state about a co-simulation, and a description of the process for generating candidate configurations for co-simulations.

2.1 The Configuration Problem

As described in [18] and explored in-depth in the thesis of Gomes [14], it can be challenging to configure a co-simulation which satisfies properties to the same degree as the original system. This is due to the inherent approximation of the original system's *behaviour trace* by the co-simulation's behaviour trace, and due to the many possible manners in which a co-simulation can be computed.

Assuming that co-simulation units are correctly implemented, the configuration problem can be stated as: *given a set of co-simulation units and their interconnections, and a set of properties that the coupled system should satisfy, find the master algorithm that produces co-simulation results satisfying those properties* (Problem 1 in [18]).

Since the correct co-simulation configuration is not available as our reference oracle, we are forced to rely on the user's hints as a proxy for the correct set of properties to satisfy. This assumption of hint correctness is quite strong, but allows us to guide the search for a correct co-simulation configuration based on a mapping from these hints to state-of-the-art configuration techniques.

2.2 Workflow

HintCO has four main components, briefly introduced here as part of the HintCO workflow:

a) HintCO Hint Language The user first specifies hints about the system by selecting and configuring built-in hints relevant to the domain of the units involved.

b) Generation of Candidate Master Algorithms HintCO automatically generates candidate co-simulation configurations based on those hints. This is performed by translating those hints into adaptations on the configurations, using state-of-the-art heuristics.

c) Scheduling the Co-simulation A co-simulation schedule is then generated for each candidate configuration using the techniques described in this paper.

d) Execution of the Co-simulation Finally, the co-simulation variants are executed in a ranked order (as determined by the hints) and the results presented to the user for inspection.

2.3 Hints

Hints are defined in a small *domain-specific language* (DSL). DSLs allow experts in the problem space (the system engineers) to describe hints, without having to become experts in the solution space (the co-simulation domain) [29].

As an example, Figure 1 shows the hints exemplified in [18]. The first hint specifies the frequency of a unit, which is useful to find a communication rate between units, and determines whether a unit represents a time triggered software controller. The second hint defines a *power-bond*, which declares that energy should not be lost or gained in the communication between two units.

```
Hint ExecRate{
    description "Controller FMU is software."
    statements {
        Property ExecRate :=
            FMUProperty FMU1.exec_rate == val 1.0e+6 hz
    }
    scope Globally
    pattern Universality:always-the-case-that ExecRate holds
}
Hint PowerBond{
    description "Plant/Load FMUs share a power connection."
    statements {
        Property PowerBond :=
            Plant.f == PowerBondSuggestions with Load.v
    }
    scope Globally
    pattern Universality:always-the-case-that PowerBond holds
}
```

Fig. 1: The *ExecRate* and *PowerBond* hints.

Each hint has a number of fields. The *description* field is for unstructured text, as is commonly seen in industrial requirements. Following this are *statements*, which can be *events* or *properties*. Finally, *scopes* and *patterns* specify when the hint is valid. These scopes and patterns have been sourced from [2] and have been utilized for verification of safety-critical automotive requirements in another domain of our work [6].

2.4 Generating a Configuration Search Space

As will be detailed later in Definition 7, a co-simulation configuration (or *master algorithm*) has three dimensions in our formalization:

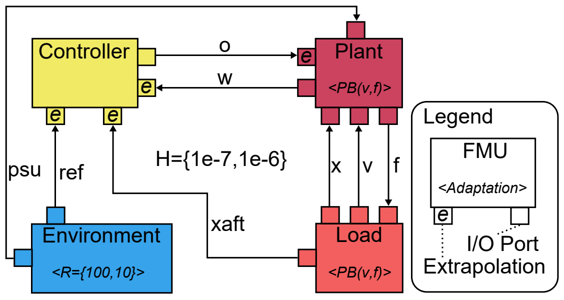
– the rate at which co-simulation units execute;
– the concrete operations or execution order of those units;
– the semantic adaptations (described below) applied to the co-simulation units.

It is not feasible to explore all possible configurations, so the hints are used as a way to build a finite ranked list of possible candidate configurations.
Semantic adaptations are a technique to create a new co-simulation unit by wrapping the old units [17] (also see Definition 4), thereby changing it's behavior when inputs are provided, when output are requested, or when time stepping is performed. Some example semantic adaptations are:

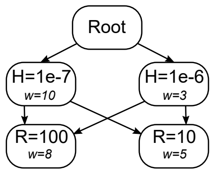
Extrapolation/Interpolation Applies the approximation to the input port.
Multi-rate Adapt a co-simulation unit to perform multiple executions per one larger co-simulation step.
Power-bond Whenever two units share a power connection, one of the input ports will correct for the energy dissipated using the technique from [4].
XOR Is combined with other adaptations to represent alternative configurations.

The following exemplifies a search space.

Example 1. Figure 2a shows the configuration space used in [18]. Possible adaptations are shown on each co-simulation unit and port: the `Load` and `Plant` co-simulation units have a *PowerBond* adaptation on the v and f ports, and the `Environment` unit has an *XorAdaptation* with two alternative multi-rate adaptations.
This space represents four alternative master algorithms, because of the two possible rates for the `Environment` unit. These adaptations are represented in the figure as the execution rate $R = \{100, 10\}$, and two possible communication step sizes $H = \left\{1 \times 10^{-7}, 1 \times 10^{-6}\right\}$.



(a) Possible adaptations.



(b) Variant diagram.

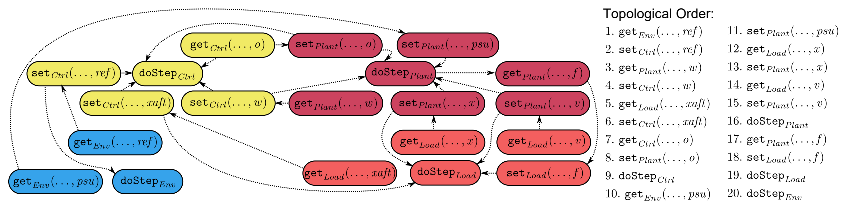
Fig. 2: Case study introduced in [18].

The key operation of the HintCO framework is to transform the user's hints into semantic adaptations, as exemplified in Figure 2a. This is described in [18, Procedure 1].

Having a configuration space, HintCO then generates a *variant diagram*, as exemplified in Figure 2b. This diagram reflects the weighting of the variants as defined by the hints, and each path from root to leave node represents a co-simulation configuration (a variant). HintCO employs a weighted depth-first search to traverse this tree and generate the variants for scheduling and execution.
For example, in Figure 2b, the search will first select the adaptations of $H = 1 \times 10^{-7}$ for the step-size and then $R = 100$ for the `Environment` unit rate, due to the highest weight. The user can opt for generate all variants, or only the top n.

2.5 Scheduling and Execution

To properly configure a co-simulation, HintCO must take a variant, and define a concrete *operation schedule* for how the co-simulation units in the scenario are executed and how values are passed between units. Operation schedules and their creation are further explored in Section 4.2. This schedule must comply with the requirements imposed by the adaptations chosen in the variant. These requirements refer to the order in which: inputs can be set; outputs computed; and time advancement operations performed. Thus, the schedule can be different for different variants.

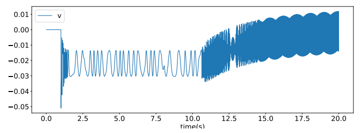

$\texttt{get}_{Ctrl}(\ldots, o)$
$\texttt{set}_{Plant}(\ldots, o)$
$\texttt{set}_{Plant}(\ldots, psu)$
$\texttt{set}_{Ctrl}(\ldots, ref)$
$\texttt{doStep}_{Ctrl}$
$\texttt{doStep}_{Plant}$
$\texttt{get}_{Plant}(\ldots, f)$
$\texttt{set}_{Ctrl}(\ldots, xaft)$
$\texttt{set}_{Ctrl}(\ldots, w)$
$\texttt{get}_{Plant}(\ldots, w)$
$\texttt{set}_{Plant}(\ldots, x)$
$\texttt{set}_{Plant}(\ldots, v)$
$\texttt{get}_{Env}(\ldots, ref)$
$\texttt{get}_{Load}(\ldots, x)$
$\texttt{get}_{Load}(\ldots, v)$
$\texttt{get}_{Env}(\ldots, psu)$
$\texttt{doStep}_{Env}$
$\texttt{get}_{Load}(\ldots, xaft)$
$\texttt{doStep}_{Load}$
$\texttt{set}_{Load}(\ldots, f)$

Topological Order:

1. $\texttt{get}_{Env}(\ldots, ref)$	11. $\texttt{set}_{Plant}(\ldots, psu)$
2. $\texttt{set}_{Ctrl}(\ldots, ref)$	12. $\texttt{get}_{Load}(\ldots, x)$
3. $\texttt{get}_{Plant}(\ldots, w)$	13. $\texttt{set}_{Plant}(\ldots, x)$
4. $\texttt{set}_{Ctrl}(\ldots, w)$	14. $\texttt{get}_{Load}(\ldots, v)$
5. $\texttt{get}_{Load}(\ldots, xaft)$	15. $\texttt{set}_{Plant}(\ldots, v)$
6. $\texttt{set}_{Ctrl}(\ldots, xaft)$	16. $\texttt{doStep}_{Plant}$
7. $\texttt{get}_{Ctrl}(\ldots, o)$	17. $\texttt{get}_{Plant}(\ldots, f)$
8. $\texttt{set}_{Plant}(\ldots, o)$	18. $\texttt{set}_{Load}(\ldots, f)$
9. $\texttt{doStep}_{Ctrl}$	19. $\texttt{doStep}_{Load}$
10. $\texttt{get}_{Env}(\ldots, psu)$	20. $\texttt{doStep}_{Env}$

Fig. 3: Example operation schedule for the co-simulation scenario in Figure 2a [18].

Figure 3 demonstrates an example of an operation schedule for the co-simulation scenario in Figure 2a, as replicated from [18]. The left-hand side of the figure defines the dependencies of function calls, as given by the interaction between the variant's adaptations and rules defining a valid co-simulation configuration (Definition 9). The right-hand side of the figure depicts a schedule of those operations, as given by a topological ordering of the dependencies.
The execution of the variants by HintCO is performed behind-the-scenes by the tool, based on the concrete operation schedule automatically produced. The user receives the co-simulation traces for each variant and can decide whether to continue with the variant tree search or not. In this way, the variant tree generation and the scheduling process are hidden to reduce complexity for the system engineer.
An example trace for the `Load` co-simulation unit from Figure 2a is shown on the left-hand side of Figure 4 before applying the hints and adaptation. The smoothed results on the right-hand side of Figure 4 are the result of the top variant, demonstrating how HintCO can assist in configuring co-simulations.


(a) Signal before adapation

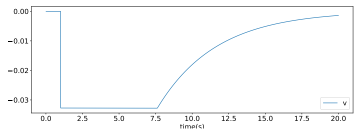

(b) Signal after adapation

Fig. 4: `Load` signal before and after HintCO adaptations are applied [18].

3 Motivating Example

This example motivates our current work regarding scheduling, as the input/output relationships of the units in the co-simulation form a dependency loop. This is termed as an *algebraic loop*.
The version of HintCO proposed in [18] could produce an operation schedule in the presence of algebraic loops, but HintCO would not consider the dynamics of each connection in deciding which dependency was the least important and could be removed to break the algebraic loop. As discussed in [19] and Section 5.3, a lower co-simulation error can be achieved by choosing a more appropriate point to break the loop.

3.1 Example Description

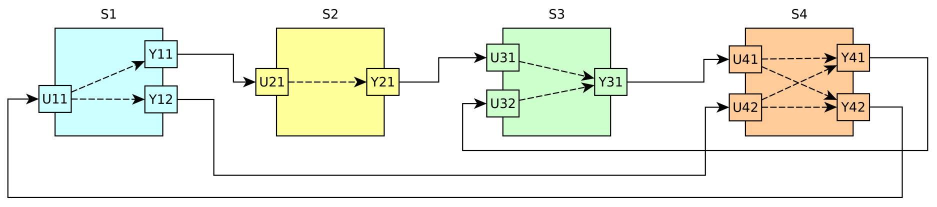


Fig. 5: Example co-simulation network with algebraic loops.

The notion which leads to algebraic loops is *feed-through* (formalized in Definition 5), where the output of a co-simulation unit algebraically depends on its inputs. That is, when the input value of a unit is modified, the connected output value immediately changes, without the unit executing.
Figure 5 shows an co-simulation scenario from [19]. Feed-through is represented in Figure 5 by the dashed arrows within the co-simulation units.
For this motivating example, we consider the case where the user does not provide sufficient hints for HintCO to break the feed-through dependency loops. If a dependency graph (such as Figure 3) was generated by HintCO, the function calls of these units would produce a graph with a cycle. Then, HintCO would have to make a decision about the best dependency to break.

In the previous version of HintCO, this decision was performed without considering the dynamics of the connections. The current work attempts to formalize this problem and present exact and heuristic solutions, such that HintCO can be improved to better schedule these co-simulation scenarios.

4 Formalization of Co-simulation Concepts

This section formalizes the key concepts involved in co-simulation configuration, in a refinement of those presented in our earlier work [18]. In particular, we recall definitions for co-simulation units and co-simulation configuration. These formalizations are required to support Section 5 which details the problem of co-simulation scheduling in the presence of algebraic loops.

4.1 Co-simulation Units and Relevant Characteristics

The following definitions focus on the concepts of a co-simulation unit and the relevant characteristics which affect co-simulation behaviour. As with our earlier formalization [18], we consider general *co-simulation units* in our scheduling approach. This notion of co-simulation unit is inspired by and includes the Functional Mockup Units (FMUs). As in our earlier work [18], we follow the notations introduced in [8] and omit the details of initialization.

Definition 1 (Co-simulation Unit). *[18, Definition 3]*
A co-simulation unit with identifier c is a structure
$\langle S_c, U_c, Y_c, R_c, \mathtt{set}_c, \mathtt{get}_c, \mathtt{doStep}_c \rangle$, *where:*

- S_c *represents the state space;*
- U_c *and* Y_c *the set of input and output variables, respectively;*
- $R_c : U_c \to \{true, false\}$ *the reactivity of each input (see Definition 3);*
- $D_c \subseteq U_c \times Y_c$ *the set of input/output feed-through dependencies (see Definition 5);*
- $\mathtt{set}_c : S_c \times U_c \times \mathcal{V} \to S_c$ *and* $\mathtt{get}_c : S_c \times Y_c \to \mathcal{V}$ *are functions to set the inputs and get the outputs, respectively (we abstract the set of values that each input/output variable can take as* $\mathcal{V}$*); and*
- $\mathtt{doStep}_c : S_c \times \mathbb{R}_{\geq 0} \to S_c$ *is a function that instructs the co-simulation unit to compute its state after a given time duration.*

When configuring co-simulations, it is crucial to reason about the current time of each co-simulation unit. The following definition defines the *state timestamp* for each unit, which the FMI Standard leaves implicit.

Definition 2 (State Timestamp). *[18, Definition 4]*
Given a communication step size $H \in \mathbb{R}_{\geq 0}$ *and* $H > 0$, *we say that the state* $s_c \in S_c$ *of an co-simulation unit c has timestamp t, denoted as* $\varphi(s_c) = t$ *when* $\mathtt{doStep}_c$ *has been called* $\frac{t}{H}$ *times with H as parameter.*

Definition 2 implies that if a co-simulation unit is in state s_c at time t, then $\mathtt{doStep}_c(s_c, H)$ will approximate the state at time $t + H$. If the corresponding model is a continuous one, then an approximation function will be used to estimate the values of the inputs in the interval $[t, t + H]$.

Input Approximation Functions There are two relevant approximation functions we focus on. There can be an *extrapolation* on an input, where the value of a input is calculated forward from the last received value. Otherwise there can be an *interpolation*, where an intermediate value is calculated between the last received value and a value from the sending unit at either the current timestamp, or a (relative) future timestamp.
Assumptions can be made about relating these these approximation functions to the behaviour of co-simulation units. For example, as mentioned in Section 2 the user can provide a hint that an co-simulation unit represents a software controller. As software controllers rely on data from sampled sensors, the software controllers assume that their input readings are not from a future timestamp. Therefore, it can be inferred that a software controller is using an extrapolation approximation function.
As interpolations can only be employed when the sending unit can already calculated the value, the choice of scheduling of co-simulation units can also impact which approximation can be used. The interpolation/extrapolation choice also affects the error of the system, as discussed in Section 5.3. Extrapolations can induce more error in the co-simulation [19], but can be employed to break algebraic loops as the dependency between units in the same timestep is then removed.
In our notation, we choose to leave the approximation function implicit in the $\texttt{doStep}_c$, as reflected in version 2.0 of the FMI Standard. However, we make explicit the requirements of each kind of input approximation in the form of the reactivity R_c.
Intuitively, a co-simulation unit c with a reactive input [16] must wait until the co-simulation unit d, which feeds that input to c, executes a step. Then, c may receive that input value.

Definition 3 (Reactivity). *[18, Definition 5]*
For a given co-simulation unit c with input $u \in U_c$, $R_c(u) = true$ if the function $\texttt{doStep}_c$ makes use of an interpolation of input u.
Let t be the timestamp of the state s_c prior to a call to $\texttt{doStep}_c(s_c, H)$, and let d denote the co-simulation unit whose output $y \in Y_d$ is connected to u.
Then, $R_c(u) = true$ means that s_c must have been produced from a call to $\texttt{set}_c\big(\ldots, u, \texttt{get}_d(s_d, y)\big)$ where the state s_d of co-simulation unit d satisfies $\varphi(s_d) = t + H$.
Conversely, $R_c(u) = false$ means that s_c must have been produced from a call to $\texttt{set}_c\big(\ldots, u, \texttt{get}_d(s_d, y)\big)$ where $\varphi(s_d) = t$.

As the FMI Standard version 2.0 does not include information about reactivity, we make the following assumption for all co-simulation units:

Assumption 1 *If an co-simulation unit c does not disclose its input approximation scheme for an input u, then we assume that u is approximated with a constant extrapolation. Therefore, $R_c(u) = false$.*

We employ the technique of *semantic adaptation*, introduced in Section 2.4 to control the input approximation scheme and reactivity for co-simulation units.

Definition 4 (Semantic Adapation). *[18, Definition 2]*
Semantic adaptation is a technique that allows a new co-simulation unit c to be constructed from a set of co-simulation units, using a custom implementation of the $\texttt{set}_c$,$\texttt{get}_c$, and $\texttt{doStep}_c$ functions [17].

Feed-through The concept of *feed-through* is crucial for the current work. If a co-simulation unit has feed-through, then an output value of a co-simulation unit is a function of the input.

Definition 5 (Feed-through). *The input $u \in U_c$ of co-simulation unit c feeds-through to output $y \in Y_c$, that is, $(u,y) \in D_c$, when there exists two values $v_1, v_2 \in \mathcal{V}$ and some state $s_c \in S_c$, such that*

$$\mathtt{get}_c(\mathtt{set}_c(s_c,u,v_1),y) \neq \mathtt{get}_c(\mathtt{set}_c(s_c,u,v_2),y).$$

This means that the value of y obtained with $\mathtt{get}_c$ is an algebraic function of the value set for u. Hence, $\mathtt{get}_c$ should be called to get the value of y only after $\mathtt{set}_c$ has been called to set the value of u, before $\mathtt{doStep}_c$ is invoked.

Co-simulation units with feed-through will immediately change output values when the corresponding input value changes, even without the co-simulation unit executing a time-step. It is this ‘instant’ change which can form algebraic loops when multiple co-simulation units have feed-through, as in the motivating example in Section 3.

4.2 Co-simulation Scenario and Master Algorithms

A *co-simulation scenario* is a collection of co-simulation units and the input/output connections between them. An example of a co-simulation scenario with four co-simulation units and six connections is shown in Figure 5 on page 7.

Definition 6 (Co-simulation Scenario). *[18, Definition 7]*
A co-simulation scenario is a structure $\langle C, L\rangle$ where each co-simulation unit identifier $c \in C$ is associated with an co-simulation unit, as defined in Definition 1, and $L(u) = y$ means that the output y is connected to input u. Let $U = \bigcup_{c\in C} U_c$ and $Y = \bigcup_{c\in C} Y_c$, then $L : U \to Y$.

Master Algorithms A *master algorithm* is the configuration to compute the behaviour of a co-simulation scenario. As stated in Definition 7, a master algorithm combines the co-simulation scenario, the step size, and a scheduling sequence for the scenario. As described previously in Section 2.4, our approach is to guide a search through the variation of these parameters, which each induce a different co-simulation behaviour.

Definition 7 (Master Algorithm). *[18, Definition 10]*
Given a co-simulation scenario $\langle C, L\rangle$, a step size H, and an operation schedule $(f)_{i\in\mathbb{N}}$, a master algorithm is a structure defined as $\mathcal{A} = \left\langle C, L, H, (f)_{i\in\mathbb{N}}\right\rangle$.

Co-simulation Step The execution of a master algorithm $\mathcal{A} = \left\langle C, L, H, (f)_{i\in\mathbb{N}}\right\rangle$ is thus the application of the operation schedule on the co-simulation scenario. One application of this sequence is a *co-simulation step*. Precisely, executing this schedule in a co-simulation scenario $\langle C, L\rangle$ where all co-simulation units $c \in C$ have a state s_c satisfying $\varphi(s_c) = t$, will update each co-simulation unit’s state s_c to satisfy $\varphi(s_c) = t + H$, where H is the argument of every call to `doStep`. Repeated co-simulation steps thus advance the state of the co-simulation scenario, producing a co-simulation behaviour trace.

Readers may note that the definition of operation schedule provided in Definition 8 was referred to as a *co-simulation step* in [18]. This redefinition was performed in order to allow the co-simulation step concept to also refer to the execution of a *trigger sequence* of co-simulation units, which is further described in Section 5.1.
This presentation omits the handling of hierarchical co-simulation units, which themselves contain a co-simulation scenario. However, this is accounted for in our treatment, as we consider the sub-scenario to execute whenever the hierarchical co-simulation unit itself executes. Thus, we create the schedule for the top-level elements separately from the schedule for each individual hierarchical co-simulation unit.

Operation Schedules In Definition 7 a master algorithm contains an *operation schedule*. This operation schedule represents the sequence of function execution for each co-simulation unit in the scenario. That is, the sequence of operations `get`, `set`, `doStep`) which are executed on each port of each co-simulation unit. We formalize this *operation schedule* concept in Definition 8.

Definition 8 (Operation Schedule). *[Modified from Definition 8 of [18]] Given a co-simulation scenario $\langle C, L\rangle$, an operation schedule is an ordered sequence of co-simulation unit function calls $(f)_{i\in\mathbb{N}}$ with*

$$f \in F = \bigcup_{c\in C} \{\texttt{set}_c, \texttt{get}_c, \texttt{doStep}_c\},$$

and i denoting the order of the function call. A function call f_i comes before a function call f_j, written as $f_i \twoheadrightarrow f_j$, if $i < j$, and comes immediately before, written as $f_i \rightarrow f_j$, if $i = j-1$.

Definition 9 states the restrictions on a well-formed master algorithm, and how the hints provided to HintCO affect this schedule. For example, if an input is reactive (it performs an interpolation approximation - Definition 3), then that input's `get` step must occur after the `doStep` of the preceding co-simulation unit.

Definition 9 (Valid Master Algorithm). *[Modified from Definition 9 of [18]] A master algorithm is valid with respect to reactivity and the co-simulation scenario couplings if it satisfies the following conditions:*

1. *A co-simulation step size $H > 0$.*
2. *Each function call uses the most recent co-simulation unit state as parameter. For example, if $f_j = \texttt{get}_c(s_c, y)$ then s_c must be the result of the most recent call to $\texttt{set}_c$ or $\texttt{doStep}_c$, that is, the maximal i such that $i < j$, and $f_i = \texttt{set}_c(\ldots)$ or $f_i = \texttt{doStep}_c(\ldots)$.*
3. *For every $c \in C$, there exists one, and only one, call to $\texttt{doStep}_c$, and it is done with argument H.*
4. *Each call to $\texttt{doStep}_c$ for $c \in C$ must come after every call to $\texttt{set}_c$ on the input variables of c.*
5. *Each call to `get` is immediately followed by a sequence of calls to `set` to set the affected input variables.*
6. *For each $c \in C$ and $(u, y) \in D_c$, any call to $\texttt{get}_c(y, \ldots)$ must be preceded by a call to $\texttt{set}_c(\ldots, u)$ without any call to $\texttt{doStep}_c$ in between.*

7. *For each $c \in C$ and $u \in U_c$ satisfying $R_c(u) = true$,* $\texttt{doStep}_d \twoheadrightarrow \texttt{get}_d(L(u), \ldots)$, *where $L(u) \in Y_d$ and $d \in C$.*
8. *For each $c \in C$ and $u \in U_c$ satisfying $R_c(u) = false$,* $\texttt{set}_c(\ldots, u) \twoheadrightarrow \texttt{doStep}_d$, *where $L(u) \in Y_d$ and $d \in C$.*

Remark 1. Regarding Definition 9:

- The most common master algorithms will satisfy conditions 2–4;
- Condition 5 is not mandatory but it facilitates the description of Conditions 7 and 8. Furthermore, it makes the implementation simpler.
- Conditions 7 and 8 ensure that the reactivity of each input is respected, according to Definition 3.
  - If $R_c(u) = true$, then the input approximation is *interpolated*, and the preceding co-simulation unit must perform `doStep` before the `get` call
  - If $R_c(u) = false$, then the input approximation is *extrapolated*, and the `set` call is performed before the preceding co-simulation unit performs `doStep`

Generating an Operation Schedule A particular variant co-simulation configuration (discussed in Section 2.4) could define interpolation or extrapolation adaptations on co-simulation units or their input ports. These adaptations interact with the rules defined in Definition 9, which specify the dependencies between the function calls in the operation schedule. This then induces a dependency graph of the function calls in the co-simulation scenario. An example of these ordering dependencies is demonstrated on the left-hand side of Figure 3 on page 3. In this dependency graph, the operation at the tail of an edge must be executed before the operation at the head of an edge.
The dependency graph in Figure 3 does not contain any cycles, due to the lack of feed-through in the co-simulation scenario (shown in Figure 2a). A topological sort is therefore sufficient to generate an *operation schedule* to execute the co-simulation units, as seen on the right-hand side of Figure 3.
This operation schedule approach allows for a great deal of flexibility in the concrete order of operations, as all topological orderings are considered to be behaviourally equivalent. A Prolog-based algorithm for specifying co-simulation scenarios and determining a valid operation schedule is presented in [12].
However, if cycles are present in this dependency graph (as in the motivating example in Section 3), the cycle will need to be broken to produce an operation schedule. As this cycle breaking produces errors (due to the extrapolation approximation used), an optimization approach is required to determine the best scheduling, as discussed in the following sections.

5 Scheduling with Algebraic Loops

This section will detail our approach to scheduling co-simulation scenarios with algebraic loops. This approach is based on co-simulation *trigger sequences*, which are an intuitive scheduling of co-simulation units at a high level. We also present how trigger sequence can be transformed into operation schedules to be executed by HintCO. Third, the cost function for a trigger sequence is defined, determined by the connections within the co-simulation scenario. Finally, we discuss a directed search approach to calculating the optimal trigger sequence.

5.1 Trigger Sequences

As described in Section 4.2, execution of a co-simulation scenario requires the production of an operation schedule, which details the precise sequence of function calls within the scenario. However, another (possibly more intuitive and more elegant) approach is to define a *trigger sequence* for the co-simulation scenario, as is done in [19]. Following this trigger sequence, each co-simulation unit would be visited and executed in turn, with input and output approximation and propagation handled as required.
The motivation for defining and utilizing trigger sequences is therefore to consider co-simulation scenarios at an abstract level. The presence of algebraic loops leads to a 'strong component' notion, where co-simulation units must be reasoned about as one entity. In particular, the problem statement in Section 5.3 deals with co-simulation units, not their individual function calls.

Example and Definition Consider the motivating example co-simulation scenario in Figure 5, which contains four co-simulation units S1, S2, S3, and S4. There are 24 different permutations of these units, and therefore 24 possible trigger sequences (Definition 10) can be created, such as {S1, S2, S3, S4} or {S2, S4, S3, S1}.

Definition 10 (Trigger Sequence). *Given a co-simulation scenario $\langle C, L\rangle$, trigger sequence is an ordered sequence of co-simulation unit executions (c_i) with $c \in C$ and i denoting the order of the co-simulation unit execution.*

5.2 Transformation to Operation Schedule

The definition of a master algorithm in Definition 7 contains a schedule of function calls on the co-simulation units. Therefore, a transformation from a trigger sequence to an operation schedule is required for co-simulation execution.
This transformation must respect the constraints defined in Definition 9 for a valid master algorithm. In particular, condition 5 must be followed, in which each call to `get` an output is immediately followed by a call to `set` for the associated input. A trigger sequence is thus transformed into an operation schedule by Procedure 1.

Procedure 1 *Produce an operation schedule from a given trigger sequence:*
- *For each co-simulation unit c in the trigger sequence:*
  - *Add the appropriate* `get` *and* `set` *calls for each input of c to the operation schedule.*
  - *Add extrapolation adaptations to co-simulation unit inputs where required (see Section 5.3)*
  - *Add the* `doStep` *call for c to the operation schedule.*

For example, Example 2 presents the operation schedule for the trigger sequence {S2, S1, S3, S4} for the co-simulation scenario in Figure 5.

Example 2. $\{\texttt{get}_{Y11}, \texttt{set}_{U21}, \texttt{doStep}_{S2}, \texttt{get}_{Y42}, \texttt{set}_{U11}, \texttt{doStep}_{S1},$
$\texttt{get}_{Y21}, \texttt{set}_{U31}, \texttt{get}_{Y41}, \texttt{set}_{U32}, \texttt{doStep}_{S3},$
$\texttt{get}_{Y31}, \texttt{set}_{U41}, \texttt{get}_{Y12}, \texttt{set}_{U42}, \texttt{doStep}_{S4}\}$

5.3 Problem Statement

As presented in Section 3, our motivating example contains feed-through of inputs and outputs arranged in a cycle. This implies that the dependency graph for operations (as in Figure 3) would also have a cycle.
Section 5.4 will answer the important question of how the above trigger sequence {S2, S1, S3, S4} was created for this co-simulation scenario, despite the cyclic dependency. However, we first focus on what impact the breaking of the algebraic loop has on our co-simulation results.
The scheduling of the co-simulation trigger sequence can impact the results of the co-simulation, as seen in related work [19]. This is due to the presence of input approximation algorithms used in the co-simulation, as discussed in Section 4.1. Recall that interpolation algorithms may interpolate input values between the previous time step and the next one. This (may) reduce error compared to extrapolations, but interpolations are only available to use on a co-simulation unit when the preceding unit has already executed. That is, any co-simulation unit c_i in a trigger sequence can interpolate values from co-simulation units $c_{j<i}$, and must extrapolate the values from co-simulation units $c_{k>i}$.
Holzinger and Benedikt [19] take these considerations into account and produce a trigger sequence which minimizes the number of extrapolations performed to reduce error. However, their technique is based on a Travelling Salesman Problem approach, which can be computationally expensive. Instead, Section 5.4 presents a directed search algorithm, and Section 6 explores heuristic algorithms to perform this scheduling.
Therefore, the problem statement considered in the following sections is: *given a co-simulation scenario, what is the trigger sequence with a minimum cost, where this cost represents performing extrapolations of co-simulation unit inputs?*
The following sections investigate trigger sequence creation and define this cost function.

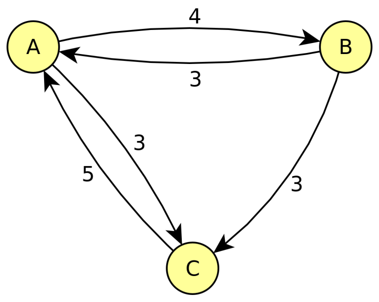
5.4 Trigger Sequence Creation

This section will detail how a trigger sequence can be constructed from a co-simulation scenario. First, we describe a *scenario graph* to intuitively represent the co-simulation units in a scenario and a measure of the dependency strength of their connections. Second, we detail how this graph is used to build a trigger sequence. Finally, a directed search approach for finding an optimal trigger sequence is presented.

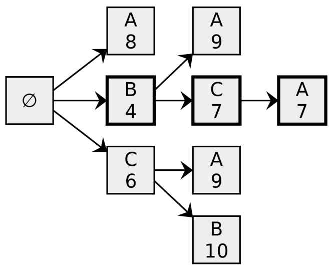
Scenario Graph A *scenario graph* represents the necessary information from a co-simulation scenario to define a trigger sequence. Vertices in this scenario graph represent each co-simulation unit in the scenario. Edges are weighted, directed, and connect vertices which are connected in the original co-simulation scenario. An example co-simulation scenario graph is shown in Figure 6a. Not shown in this example is that each edge in this scenario graph could represent more than one co-simulation connection in the original scenario. The weight of the edges represent a measure of the interdependence of the units in the original scenario, such as a count of the number of connections. More advanced analysis are possible to represent a more nuanced calculation of sensitivity of the connection, as in [19].

In our formulation of the scenario graph, which follows [19], the weight of each edge represents the cost for performing an extrapolation approximation on the input/output connections in the original co-simulation scenario, which are represented by that edge. This weight can be set through hints from the user, though we are exploring automated determination of costs.
Selecting the weight of an edge must also take into account any interpolation or extrapolation information provided by hints on the scenario. That is, the variant generation described in Section 2 may determine the approximation for certain co-simulation unit inputs. This sets the weight of the relevant edge to zero, as the hint suggests that the co-simulation unit is constructed to appropriately handle the resulting approximation error.

Trigger Sequence Cost Function Following the definition of the co-simulation graph above, building a trigger sequence involves selection of the co-simulation units to execute, taking into account the weights of the edges between them.
For example, consider the situation where the A unit in Figure 6a is executed first in a trigger sequence. Both the inputs from B and the inputs from C must be extrapolated, for a summed cost of eight.



(a) Scenario graph.



(b) Directed search tree.

Fig. 6: An example scenario graph and a directed search tree for the optimal trigger sequence.

The real complication in determining the optimal trigger sequence arises in that scheduling the execution of a co-simulation sets the costs of outgoing edges of that unit to zero. That is, co-simulation units on outgoing edges will not be forced to extrapolate the output, but instead can rely on interpolation, which is beneficial to the error of co-simulation [19].
As an example, consider the co-simulation graph in Figure 6a. Each of the six possible trigger sequences for the co-simulation scenario has a different cost given by this interpretation. If co-simulation unit B is selected for execution in the trigger sequence first, the cost will be four due to the weight of the edge from A. However, the weight on the edge from B to C will then be zero, as C can now interpolate the output value of B.

Cost Equation The following equation defines the cost function for a trigger sequence, in a reformulation of the cost equation found in [19]. Informally, each node is considered and incoming weights from nodes not yet encountered in the trigger sequence are summed.

Let $G = (V,E)$ denote the scenario graph, where $E \subseteq V \times V \times \mathbb{R}$ denotes the (positively-)weighted edge set. Given a trigger sequence $v_0, v_1, \ldots, v_n$, with distinct v's, $n = |V|$, and $v_i \in V$ for all $i = 0, \ldots, n$, its cost is:

$$\begin{aligned} c(v_0, v_1, \ldots, v_n) &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} w(v_j, v_i) \\ w(u,v) &= \begin{cases} x \text{ if } (u,v,x) \in E \\ 0 \text{ otherwise.} \end{cases} \end{aligned} \tag{1}$$

An alternative formulation to the above would be to sum up the incoming edge weights for each node, and then subtract the outgoing edge weights for those nodes not yet visited.

As an example, consider two trigger sequences $\{B,C,A\}$ and $\{A,B,C\}$ for the scenario graph in Figure 6a. The total cost for these trigger sequences is provided in Example 3.

Example 3.

Trigger Sequence: $\{B,C,A\}$	Trigger Sequence: $\{A,B,C\}$
Cost: B = 4, C = 3, A = 0	Cost: A = 8, B = 0, C = 0
Total: 7 (optimal)	Total: 8

Directed Search Given Equation (1) for defining the cost of a trigger sequence, an brute-force algorithm can be easily created: a) all possible trigger sequences are created, b) each is evaluated using Equation (1), c) the sequence with the lowest cost is the optimal one. However, this brute-force approach is computationally infeasible, as the number of possible trigger sequences is a factorial explosion of the number of co-simulation units.

A *directed search* is instead preferable to find the optimal solution. This directed search builds up a tree of possible sequences, selecting the next branch to expand based on the cost of the branch so far. This search is possible because the cost function is defined for partial trigger sequences, and is *consistent* as well. That is, adding further nodes to a trigger sequence can only maintain or raise the total cost, so there cannot be a local maxima reached in the search.

The directed search begins with the root of the tree as the empty set $\emptyset$. Then in an iterative manner the branch with the lowest cost is expanded, by adding as children all those nodes not considered yet in that branch of the tree.

For example, Figure 6b demonstrates the final search tree for the scenario graph in Figure 6a. The layer just below the root in Figure 6b considers the execution of each node individually. As partial sequence {B} has the lowest cost, it would be expanded next. Those children ({B, A} and {B, C}) have a higher cost than {C}, so the {C} branch is expanded next into {C, A} and {C, B}. The search returns to {B, C}, which is expanded into {B, C, A} (bolded in Figure 6b) which is the optimal solution with a cost of seven.

This directed search provides the optimal solution, but could be exhaustive and therefore computationally prohibitive. The next section presents the Travelling Salesman and optimal branching approaches, along with heuristic algorithms to find a trigger sequence with near-optimality but at lower computational complexity.

6 Approaches for Constructing Trigger Sequences

This section details approaches for producing a trigger sequence with the lowest cost (as defined by Equation (1)), while avoiding the computational complexity of the directed search approach described in Section 5.4. Each proposed approach will be presented along with a discussion, including counter-examples if known.

6.1 Travelling Salesman Problem

The Travelling Salesman Problem approach to trigger sequence construction is to find a walk (or Hamiltonian cycle) which visits each unit in the scenario graph once. While this approach is intuitive, as one can think of execution as 'walking' around the scenario graph, this approach is overly restrictive and computationally expensive.

First, the trigger sequence to be produced does not have to be a cycle, nor is it required that co-simulation units which are next to each other in the trigger sequence have to be directly connected in the scenario graph. For instance, consider a scenario graph with three nodes A, B, and C, where A and B are connected to each other, and B and C are connected to each other. Clearly there cannot exist a cycle that visits every unit exactly once. Therefore, Hamiltonian cycles are not required for a trigger sequence.

A second issue with the Travelling Salesman Problem approach is that the starting unit of the cycle must also be selected, which induces another optimization problem. For example, assume that a Travelling Salesman algorithm gives the optimal solution for the scenario graph in Figure 6a, which is executing B, C, and then A for a total cost of seven. However, this cannot be treated as a cycle, for while the trigger sequence $\{B,C,A\}$ has a total cost of seven, the trigger sequence $\{A,B,C\}$ has a total cost of eight due to the extrapolations required.

Based on the above discussion, Travelling Salesman Problem approaches to co-simulation scheduling are certainly intuitive but are not the correct approach.

6.2 Optimum Branching

Another approach to trigger sequence construction is to determine the *optimum branching* for a scenario graph, such as a (minimum) spanning tree. That is, which set of edges spans the entire graph with minimal cost. This approach is also highly intuitive, as the optimal solution must have the minimum weight from incoming edges for each node. However, this approach does not provide the ordering of the nodes which provides that minimal cost.

For example, consider again the scenario graph from Figure 6a. The optimal branching is the edges B→A and A→C, with a cost of six. However, it is unclear how this minimal tree relates to the optimal trigger sequence of $\{B,C,A\}$,

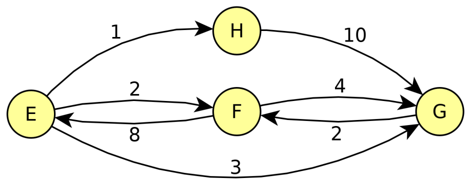


Fig. 7: Second example scenario graph.

which has a cost of seven. Therefore, the minimal spanning tree can provide a lower bound on the optimal trigger sequence cost, but (currently) cannot be used to produce this optimal trigger sequence.
From this counter-example, it is clear that while the scheduling problem is related to an optimum branching problem, it is not sufficient to apply optimal branching algorithms. This is due to the order of nodes in the trigger sequence affecting the weights of the edges, which is not the case in standard graph problems.

6.3 Greedy Approaches

As discussed in Section 5.4, a directed search is able to find the optimal trigger sequence, given enough computational resources. However, it would be beneficial to have algorithms with lower computational complexity for producing trigger sequences.
In this section, we present a number of *greedy approaches* to the trigger sequence construction problem. We note that these algorithms are not optimal and we do not provide formal bounds on the relative error to the optimum. However, our contribution is to provide a selection of algorithms that could be used for scheduling, until future work provides an optimal algorithm (if it exists). These algorithms will be evaluated in Section 7, which provides relative error results of the algorithms against the optimum on synthetic scenario graphs.
For the below algorithms, we will present examples calculations based on Figure 6a and Figure 7. These calculations are not intended to prove optimality, but only to illustrate the operation of the algorithms. As well, differences in implementation tie-breaking may also affect the results.

Lowest Incoming As a simple greedy algorithm, *Lowest Incoming* selects the node from those remaining which has the lowest incoming weights. The intuition is that the trigger sequence should be built according to choosing the 'cheapest' node next.

Algorithm:
– While not all nodes are in the trigger sequence:
  - Calculate the node with the lowest sum of incoming edge weights.
  - Add that node to the trigger sequence.
  - Remove cost for outgoing edges.

Examples

Scenario	*Found Sequence*	*Cost*	*Opt. Sequence*	*Cost*
Figure 6a	$\{B,C,A\}$	7	$\{B,C,A\}$	7
Figure 7	$\{H,F,E,G\}$	5	$\{F,E,H,G\}$	4

6.4 Benefit Ratio

The next greedy algorithm we present is *Benefit Ratio*. In this algorithm, the ratio between the incoming edge weights and the outgoing edge weights for each node is calculated such that the node with the highest 'benefit' can be selected next. The ratio is calculated as output weight divided by input weight, and nodes selected from high ratio to low.
There are two versions of this algorithm which we evaluate here. The first is *static* where the benefit ratio is determined at the beginning of the scheduling process. The second version is *dynamic*, where the benefit ratio is re-calculated after each sequence addition, to take into account that the input weights of other nodes are then reduced.

Algorithm

```
1. Calculate the benefit ratio for each node.
2. While not all nodes are in the trigger sequence.
     – Add the node with the highest benefit ratio.
     – If the dynamic version, recalculate the benefit ratios.
```

Examples for the Static Version

Scenario	*Ratios*	*Found Sequence*	*Cost*	*Opt. Sequence*	*Cost*
Figure 6a	A=0.88, B=1.5, C=0.83	$\{B,A,C\}$	9	$\{B,C,A\}$	7
Figure 7	E=0.75, F=3, G=0.12, H=10	$\{H,F,E,G\}$	5	$\{F,E,H,G\}$	4

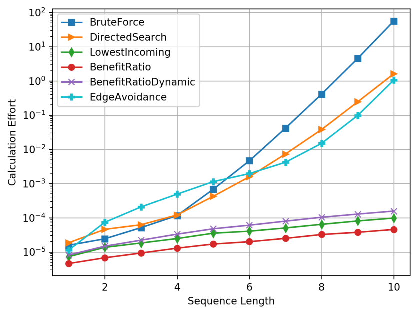
Edge Avoidance Based on the idea of optimum branching, the *Edge Avoidance* greedy algorithm has the intention of ensuring that the most expensive edges in the graph have their cost reduced to zero. The most expensive edges are selected such that maximal spanning tree is created and a topological sort gives the trigger sequence. An optimality improvement to this algorithm may be to perform a brute-force search on all possible topological sorts to find the one with the lowest cost.

Algorithm

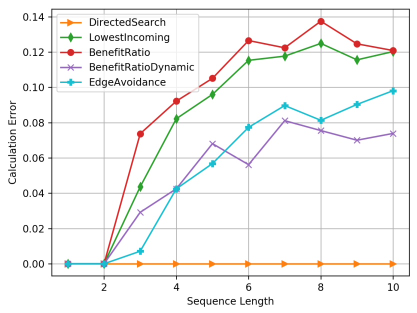
```
1. Sort the edges in descending order of weight.
2. From each edge from the beginning of that list:
     – Connect those nodes, unless doing so would create a cycle.
3. Produce the first possible topological sort.
```

Examples

Scenario	*Max. Weight Edges*	*Found Sequence*	*Cost*	*Opt. Sequence*	*Cost*
Figure 6a	C $\rightarrow$ A $\rightarrow$ B	$\{C,A,B\}$	9	$\{B,C,A\}$	7
Figure 7	H $\rightarrow$ G, F $\rightarrow$ G, F $\rightarrow$ E	Five possibilities, including optimal	-	$\{F,E,H,G\}$	4



(a) Calculation effort of each algorithm.



(b) Relative error versus the brute-force search.

Fig. 8: Evaluation of the algorithms for both effort and relative error.

7 Algorithm Evaluation

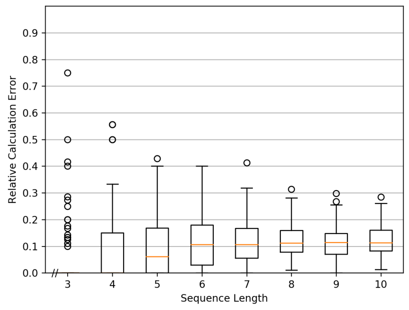
This section will provide evaluations of the brute-force, directed search, and greedy algorithms presented in Section 5.4 and Section 6. In particular, measures of relative performance and optimality are discussed to provide insight into their characteristics.

7.1 Set-up

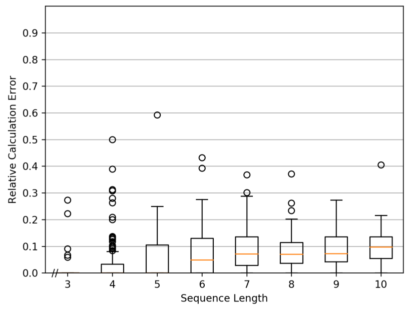
As we are unaware of a large corpus of co-simulation scenario graphs, synthetic graphs are studied here. One hundred graphs for each sequence length from one to ten were created with cycles, and edges were randomly assigned discrete weights uniformly sampled between zero and nine (inclusive).
For each graph, a brute-force search for the optimal trigger sequence is first performed to set a baseline of the performance and optimal cost. Then, the directed search (from Section 5.4) is calculated to determine the speedup given. Finally, each greedy algorithm from Section 6 is ran to determine the (potential) speedup and cost provided. The calculation effort is given in seconds, as determined by a Python 3.7.3 script running on Xubuntu 19.10 with a Intel i7-8850H CPU at 2.60GHz.

7.2 Results and Discussion

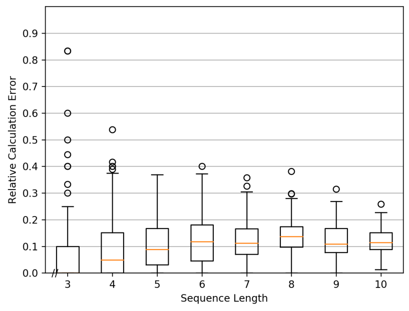
Figure 8 provides an overview of the algorithm evaluations. Figure 8a presents the average of calculation time (in a log scale) for each algorithm over all graphs of a certain size. Figure 8b presents the relative error of each algorithm versus the optimal given by the brute-force approach. The relative error is calculated by taking the difference between the cost and the optimal cost, then dividing by the optimal cost. A relative error of 0.10 therefore means the cost is 10 percent worse than the optimal. Figure 9 provides more details by presenting boxplots for each algorithm's relative error.



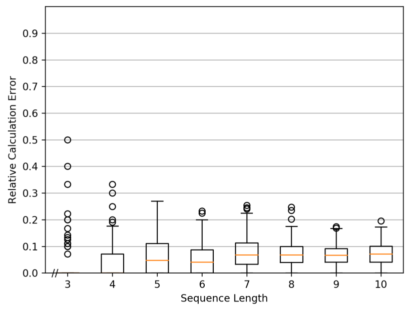
(a) *Lowest Incoming*.



(b) *Edge Avoidance*.



(c) *Static Benefit Ratio*.



(d) *Dynamic Benefit Ratio*.

Fig. 9: Detailed relative error of each algorithm.

Concerning computation time, it was expected that the directed search algorithm would be less expensive than the brute-force search. However, the results in Figure 8a show that the improvement is not as great as expected. The high cost of the *Edge Avoidance* algorithm was also unanticipated, as it is similar in cost to the *directed search*, but is not likely to always produce an optimal result. However, the other three algorithms show a definite cost improvement over the brute-force and *Directed Search* approaches.

The relative error of the algorithms versus the brute-force search shown in Figure 8b and Figure 9 provide interesting insights. The four greedy algorithms show promising results in terms of relative error. For each, the mean relative error is around 10 percent, the upper quartile falls around 25 percent, and values are rarely seen above 30 percent. As reference, a *Random Sequence* algorithm was also developed which simply makes a random decision which node to take next. For this algorithm, the mean relative error was around 40 percent, with the lower quartile at around 25 percent, and upper quartile at around 50 percent.

From the results, two algorithms are clearly superior. First, the *Directed Search* algorithm should be chosen if the optimal solution is desired and computational resources are sufficient. If a greedy approach is desired, then the *Dynamic Benefit Ratio* algorithm provides a low relative error at a low performance cost.

8 Related Work

The problem of adequately configuring a co-simulation is not new. We can classify the approaches according to when and which information is used to configure the co-simulation: static and adaptive.
An adaptive configuration approach monitors the co-simulation results and adjusts the co-simulation algorithm parameters accordingly. A static configuration approach sets the parameters without running the co-simulation. An overview of the adaptive configuration approaches is given in [18].
In the static configuration category, the following works have the same goal as our work. Rather than starting from a co-simulation scenario, the works in [5, 21, 27] use a system architecture model to generate a co-simulation scenario and a master algorithm that is consistent with that architecture. The work in [5] uses the input/output feed-through and the kind of model underlying the co-simulation unit (e.g., ODE, DAE, ...) to correctly configure the input approximations used. The authors in [27] go even further and use the eCl@ass classification system to automatically link the units.
We complement these works by showing other examples of information that are useful to configure the co-simulation, and generating multiple master algorithms, instead of a single one. This is due to the fact that there might not be enough information available to fully specify a single master algorithm.
In the domain of scheduling co-simulations,the authors of [11] provide a *sequence calculation* concept, which is analogous to our trigger sequence. Their work defines an optimization problem minimizing the communication delays between the co-simulation units in the sequence, while taking into account input/output dependencies. The optimization algorithm provides an almost optimal solution over a co-simulation scenario with 14 units. Our work instead focuses on the issue of breaking algebraic loops in the co-simulation scenario, and determining optimal and heuristic algorithms for scheduling.
The current work is based off of the approach by Holzinger and Benedikt [19] which examines the Travelling Salesman Problem (TSP) approach to scheduling co-simulations with algebraic loops. We extend that work by presenting further discussion about approaches to the scheduling problem, including a counter-example for the TSP approach. The current work also provides optimal and heuristic-based algorithms with a lower performance cost than the brute-force algorithm found in [19].

9 Conclusion

Configuration of co-simulation scenarios can be complicated due to its black-box nature, required knowledge of numerical techniques, and lack of a reference solution. In this paper, we have summarized our HintCO technique for (semi-) automatically configuring co-simulations based on user intuitions about the system, as expressed through hints.
This work has also presented our advancements in understanding the scheduling problem for co-simulations with algebraic loops. The formalization of the problem suggests that the problem is in or around NP-complexity and that it will be difficult or impossible to arrive at a polynomial-time algorithm. Second, we have determined that typical graph-based algorithms to visit all nodes such as the

Travelling Salesman Problem and Minimum Spanning Trees do not sufficiently deal with the issue that node ordering changes the cost function during visitation. As a concrete contribution, we have determined that a directed search (presented in Section 5.4) can find an optimal solution, given enough time. We have also presented heuristic algorithms to solve this problem along with results which indicate their performance and a measure of optimality.
This work is being integrated as scheduling improvements in our HintCO tool [13]. In particular, hints can now be added to co-simulation scenario connections to indicate their weight in a scenario graph. If an algebraic loop is detected in a scenario then HintCO performs the directed search for the optimal trigger sequence, then transforms it back to a dependency graph to be executed. This search and transformation is performed 'behind-the-scenes', so that the user is shielded from the complexity of co-simulation scheduling.

9.1 Future Work

One important direction for our future work is to determine whether there is an algorithm which is less computationally expensive than the directed search approach (Section 5.4) but which still guarantees optimality. We suspect that the co-simulation scheduling problem to be in the NP class, but a proof is required.
As well, we are investigating integrating additional hints into HintCO to support other scheduling problems. For example, co-simulation can be performed over a network as in [22], where co-simulation units are distributed geographically or within a network. One problem which arises could be the partitioning of co-simulation units to each network node, depending on their dependence on other nodes. This problem has been considered in a slightly different context in [20].

Acknowledgments

The authors thank Dr. Guillermo Alberto Perez (University of Antwerp) and Dr. Romain Franceschini (University of Corsica) for illuminating discussions on the trigger sequence cost function and algorithms.
This research was partially supported by a PhD fellowship grant from the Research Foundation - Flanders (File Number 1S06316N).

References

1. Arnold, M., Clauß, C., Schierz, T.: Error analysis and error estimates for co-simulation in FMI for model exchange and co-simulation v2.0. In: Progress in Differential-Algebraic Equations. pp. 107–125. Springer Berlin Heidelberg (2014). https://doi.org/10.1007/978-3-662-44926-4_6
2. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Transactions on Software Engineering **41**(7), 620–638 (2015). https://doi.org/10.1109/TSE.2015.2398877
3. Benedikt, M., Watzenig, D., Zehetner, J., Hofer, A.: Macro-step-size selection and monitoring of the coupling error for weak coupled subsystems in the frequency-domain. V International Conference on Computational Methods for Coupled Problems in Science and Engineering pp. 1–12 (2013)

4. Benedikt, M., Watzenig, D., Zehetner, J., Hofer, A.: NEPCE-A Nearly Energy Preserving Coupling Element for weak-coupled problems and co-simulation. In: IV International Conference on Computational Methods for Coupled Problems in Science and Engineering. pp. 1–12 (2013)
5. Benedikt, M., Holzinger, F.R.: Automated configuration for non-iterative co-simulation. In: 17th International Conference on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems (EuroSimE). pp. 1–7. IEEE (2016). https://doi.org/10.1109/EuroSimE.2016.7463355
6. Bernaerts, M., Oakes, B., Vanherpen, K., Aelvoet, B., Vangheluwe, H., Denil, J.: Validating industrial requirements with a contract-based approach. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 18–27. IEEE (2019). https://doi.org/10.1109/MODELS-C.2019.00010
7. Blockwitz, T., et al.: Functional Mockup Interface 2.0: The standard for tool independent exchange of simulation models. In: 9th International Modelica Conference. pp. 173–184. Linköping University Electronic Press (2012). https://doi.org/10.3384/ecp12076173
8. Broman, D., et al.: Determinate composition of FMUs for co-simulation. In: Eleventh ACM International Conference on Embedded Software. p. Article No. 2. IEEE Press Piscataway, NJ, USA (2013)
9. FMI: List of tools implementing the FMI standard, `https://fmi-standard.org/tools/`
10. FMI.: Functional Mock-up Interface for model exchange and co-simulation (2014), `https://fmi-standard.org/downloads/`
11. Glumac, S., Kovacic, Z.: Calling sequence calculation for sequential co-simulation master. In: Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. pp. 157–160. ACM (2018)
12. Gomes, C., Lúcio, L., Vangheluwe, H.: Semantics of co-simulation algorithms with simulator contracts. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 784–789. IEEE (2019)
13. Gomes, C., et al.: HintCO public repository, `https://msdl.uantwerpen.be/git/claudio/HintCO`
14. Gomes, C.: Property Preservation in Co-simulation. Ph.D. thesis, University of Antwerp (2019)
15. Gomes, C., et al.: Co-simulation: State of the art. Tech. rep., University of Antwerp (2017), `http://arxiv.org/abs/1702.00686`
16. Gomes, C., et al.: Co-simulation: A Survey. ACM Computing Surveys **51**(3), Article 49 (2018). https://doi.org/10.1145/3179993
17. Gomes, C., et al.: Semantic adaptation for FMI co-simulation with hierarchical simulators. SIMULATION **95**(3), 1–29 (2018). https://doi.org/10.1177/0037549718759775
18. Gomes, C., et al.: HintCO – Hint-based configuration of co-simulations. In: Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - Volume 1: SIMULTECH,. pp. 57–68. INSTICC, SciTePress (2019). https://doi.org/10.5220/0007830000570068
19. Holzinger, F., Benedikt, M.: Optimal trigger sequence for non-iterative co-simulation. In: Proceedings of the 9th International Conference on

Simulation and Modeling Methodologies, Technologies and Applications. pp. 80–87. SCITEPRESS - Science and Technology Publications (2019). https://doi.org/10.5220/0007833800800087

20. Holzinger, F.R., Benedikt, M.: Hierarchical coupling approach utilizing multi-objective optimization for non-iterative co-simulation. In: Proceedings of the 13th International Modelica Conference. pp. 735–740. No. 157, Linköping University Electronic Press (02 2019). https://doi.org/10.3384/ecp19157735
21. Krammer, M., Fritz, J., Karner, M.: Model-based configuration of automotive co-simulation scenarios. In: 48th Annual Simulation Symposium. pp. 155–162. Society for Computer Simulation International (2015)
22. Krammer, M., et al.: The Distributed Co-Simulation Protocol for the integration of real-time systems and simulation environments. In: Proceedings of the 50th Computer Simulation Conference. p. 1. SummerSim ‘18, Society for Computer Simulation International (07 2018). https://doi.org/10.22360/summersim.2018.scsc.001
23. Kübler, R., Schiehlen, W.: Two methods of simulator coupling. Mathematical and Computer Modelling of Dynamical Systems **6**(2), 93–113 (2000). https://doi.org/10.1076/1387-3954(200006)6:2;1-M;FT093
24. Lee, E.A.: Cyber physical systems: Design challenges. In: 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC). pp. 363–369 (2008). https://doi.org/10.1109/ISORC.2008.25
25. Schweiger, G., et al.: Functional Mock-up Interface: An empirical survey identifies research challenges and current barriers. In: Proceedings of the American Modelica Conference. pp. 138–146. Linköping University Electronic Press (2018). https://doi.org/10.3384/ecp18154138
26. Sicklinger, S., et al.: Interface Jacobian-based co-simulation. International Journal for Numerical Methods in Engineering **98**(6), 418–444 (2014). https://doi.org/10.1002/nme.4637
27. Stecken, J., Lenkenhoff, K., Kuhlenkötter, B.: Classification method for an automated linking of models in the co-simulation of production systems. Procedia CIRP **81**, 104–109 (2019). https://doi.org/10.1016/j.procir.2019.03.019
28. Van Tendeloo, Y., Vangheluwe, H.: Increasing the performance of a Discrete Event System Specification simulator by means of computational resource usage activity models. SIMULATION **93**(12), 1045–1061 (2017). https://doi.org/10.1177/0037549717726595
29. Vangheluwe, H., de Lara, J., Mosterman, P.J.: An introduction to multi-paradigm modelling and simulation. In: Proceedings of AI, Simulation and Planning in High Autonomy Systems. pp. 9–20. SCS (2002)
30. Vanherpen, K., Denil, J., De Meulenaere, P., Vangheluwe, H.: Design-space exploration in model driven engineering: An initial pattern catalogue. In: 1st International Workshop on Combining Modelling with Search and Example-Based Approaches (CMSEBA). pp. 42–51. CEUR Workshop Proceedings (Vol-1340) (2014)