Modular Design of Hybrid Languages by Explicit Modeling of Semantic Adaptation

Sadaf Mustafiz[†]
sadaf@cs.mcgill.ca

Cláudio Gomes[‡]
claudio.gomes@uantwerpen.be

Bruno Barroca[†]
bbarroca@cs.mcgill.ca

Hans Vangheluwe[‡†]
hans.vangheluwe@uantwerpen.be

[‡]University of Antwerp
[†]McGill University

ABSTRACT

The engineering of a complex cyber-physical system (CPS) involves the creation and simulation of hybrid models often encompassing multiple levels of abstraction and combining different formalisms, often not expressible in any single existing formalisms. Modular language engineering is thus essential for effective and efficient development of new formalisms, appropriate for the task.

In our work, each modeling language is represented as a language specification fragment: a modular representation of the syntax and semantics. We propose a white-box technique for explicitly modeling the definition and composition of the fragments. In this paper, we focus on the composition of the operational semantics (i.e., the semantic adaptation) of hybrid languages. This enables automatic synthesis of a simulator for the hybrid language. Our approach is demonstrated by creating two well-known hybrid languages as a composition of Timed Finite State Automata (TFSA) and Causal Block Diagrams (CBD) – hybrid TFSA and hybrid CBD.

Author Keywords

Modular language engineering; Semantic adaptation; Language composition; Hybrid languages; Fragments.

ACM Classification Keywords

I.6.0 SIMULATION AND MODELING: General; I.6.2 SIMULATION AND MODELING: Simulation Languages

1. INTRODUCTION

The modeling and simulation of hybrid models in the cyber-physical systems domain brings about many challenges. Integration – the interconnection of the components that comprise a system – is identified as a major source of problems in the concurrent development of complex systems. Such systems exhibit behavior that is best represented with a combination of continuous and discrete model constructs [19]. Hybrid models should be created using the right formalisms at the most appropriate level of abstraction [14]. To this end, we need to use a blend of formalisms to model different components. For example, in a power window system, differential equations model the power window motor and the window dynamics, whereas logical specifications (e.g., state automata) model the decision making control.

It is important to not restrict the formalisms to be used for modeling such systems in any way. This implies that new formalisms (hybrid languages) need to be developed on-demand in order to provide sufficient expressiveness. In our work, we show how language engineering techniques can be applied to develop new formalisms by reusing existing ones in a modular fashion.

The creation of new hybrid languages specifically tailored for certain domains involves not only composition of the syntax but also the semantics, including defining the extra concepts that represent the interactions between the formalisms composed (e.g., the threshold crossing condition in the Hybrid language, Section 2.1) [8]. In addition to having a well defined meaning, these interactions also need to be valid with respect to the interactions of the components in the real system. Moreover, in the context of operational semantics, the hybrid languages must satisfy several soundness properties: language continuity, determinism, time progression, time synchronization, and fairness (Section 6). All these requirements make the composition of the operational semantics (semantic adaptation) a challenging and error prone task, even for the simplest of formalisms, such as Finite State Automata (FSA) or Causal Block Diagrams (CBD). That is why, in our framework, we aim at making the semantic adaptations independent of the simulators involved, such that they can be reused for other compositions (Section 5).

We take inspiration from the initial ideas proposed in [11, 8] for the composition of languages. In this paper, we focus on re-using and composing semantics given by existing simulators, for the purpose of developing hybrid languages (Section 4). We take both discrete-event simulators (DES) and continuous-time simulators (CTS) into account, and use languages that are representative of the two domains: Finite State Automata for DES and Causal Block Diagrams for CTS (Section 2).

We use the concept of Language Specification Fragments (LSF) earlier proposed in [15]. Such fragments include the syntax (both abstract and concrete), the semantics, and the user-interface behavior, along with various interleavings of the elements involved (Section 3). The LSFs form the basis of reuse and abstraction in language design, and reduce the effort in the development of new formalisms by re-using and composing existing fragments.

In [15], we presented our work on the composition of the abstract syntax and semantics of languages with the use of a single case study (CTS in DES). The work was at an initial stage and the composition of the operational semantics (otherwise referred to as *semantic adaptation*) was shown for a particular instance. In this paper, we extend our scope by working with both kinds of composition (DES in CTS and CTS in DES). We propose a generic structure to model all DES and CTS simulators, and explicitly model the semantic adaptation required to compose these simulators using techniques that form the basis of generalizing and automating the composition process.

2. CASE STUDIES

This section introduces the well-known hybrid languages that we use to demonstrate the language composition. We begin by describing the elementary languages used in the case studies. Timed Finite State Automata (**TFSA**) is a timed variant of Finite State Automata (FSA) with a single clock [1]. A TFSA consists of a set of states and transitions between them, that (optionally) include triggers. Causal Block Diagrams (**CBD**) is a modeling language commonly used for embedded control design that models systems with differential equations. CBD is the basic language of Mathworks Simulink®. It consists of blocks and connections between blocks. Without loss of generality, we assume here that each block in a CBD has only one output but can have multiple inputs. The block itself can either represent an algebraic operation or a time sensitive operation (e.g., derivative or integral of the input). The simulation of a CBD is carried out by the following steps: 1. sort the blocks topologically according to their dependencies; 2. for each block b in the sorted list, compute its output at time t according to the kind of block, and copy it to the input of the blocks that depend upon b; 3. increment the time variable $t := t + h$.

Hybrid languages based on the TFSA and CBD can be developed by composing the TFSA and CBD in different ways: 1. TFSA composed of a CBD (referred to as **hybrid TFSA**); 2. CBD composed of a TFSA (referred to as **hybrid CBD**).

2.1 Hybrid TFSA

The Hybrid TFSA is similar to the well known Hybrid Automata formalism [9]. This formalism is used to model systems that can undergo *very fast* changes in governing dynamics, by abstracting those changes as being instantaneous. A Hybrid TFSA model – similarly to a TFSA model – is composed of a set of discrete states (also known as modes) and transitions between them. Each discrete state is associated with a CBD model that describes the dynamics when the system is in that state. The guards in the transitions describe in which conditions the system changes discrete state: **always** – as soon as it enters the source discrete state; **after** h – after h units of time since it entered the source discrete state; **when** x**+-** – when some state variable x crosses the zero threshold

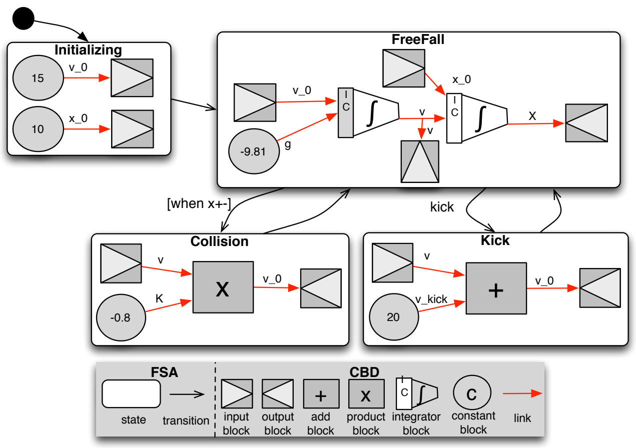


Figure 1. Bouncing Ball Hybrid TFSA Model.

from the positive to the negative direction. We also considered the when x-+, which is analogous.

The bouncing ball is an example of a Hybrid TFSA model. The system can be abstracted as having three main discrete states: 1. free falling; 2. colliding with the floor; 3. being kicked; The dynamics change when the ball collides with the ground and bounces up again with reduced energy. To illustrate the use of external events, we added a `kick` event. Figure 1 shows the complete model. In a typical scenario, the ball falls with an initial velocity $v_0 = 15$ and at an initial height $x_0 = 10$. During the free fall, gravity acts on the ball until it collides with the floor (x_0 crosses 0). At that instant, a collision occurs and the new v_0 is given by $v_0 = -0.8v$. Still in the same instant, the ball goes back to free falling again. The `Collision` discrete state is mythical [13]. Notice that the CBDs in the states have input and output ports. It is through these ports that values are propagated across multiple discrete states. When the ball goes back to free falling, the v_0 is used as the new initial velocity and x_0 remains the same. The process is similar when the ball is kicked, except that, for illustrative purposes, we assume a constant increment in the velocity of the ball. The simulation results of the bouncing ball are shown in Figure 3a. The last plot of Figure 3a highlights the fact that the CBD simulator time is different than the TFSA simulator time (see Section 4.1).

2.2 Hybrid CBD

The Hybrid CBD language is similar - albeit restricted in expressiveness for simplicity - to the composition of Stateflow models within a Simulink model. The general idea is to use a TFSA model to specify the behavior of a CBD Block. A Hybrid CBD model is composed of blocks and connections between blocks. A block can be a TFSA model and the transitions it contains can be guarded with inequalities referring to the inputs of the block. Notice that this is different than `when` transitions of the Hybrid FSA language as the latter behaves as an event occurring when the signal crosses zero (thus making use of the previous and current values of the signal), whereas the inequality does not detect a crossing.

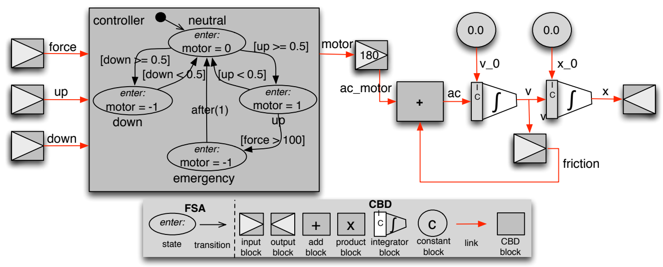


Figure 2. Power Window Hybrid CBD Model.

A power window [18] is a well-known example of a hybrid CBD model (Figure 2, adapted from [5]). The `up` and `down` signals represent the pressing of the up/down buttons of the window interface. The `force` signal comes from a sensor measuring obstruction force applied to the window. The block with the TFSA model handles the decision-making by monitoring the input signals and changing to the appropriate state. Each state is associated with a value that represents the output of the block when the state is active. In a typical scenario, the `up` signal is greater than 0.5 so the controller block goes to state *up*, setting the motor signal to 1. If the `force` signal exceeds 100, the controller goes to *emergency*, reversing the motor signal for 1 second. The plots in Figure 3b show the behavior of the power window system model. Although the example only includes one TFSA, we worked with more complex models that include multiple blocks with TFSAs.

3. FRAGMENTS

Language fragments can be defined as reusable components in the language engineering process. We define a language specification fragment (LSF) as a specification of a modeling language in terms of its abstract syntax (AS), concrete syntax (CS), operational semantics (OS), and the user-interface (UI) behavior. As illustrated in Figure 4, and detailed in [15], such specification involves the definition of several mapping models (referred to as m1-m9). We focus on the OS – the composition of which allows the generation of a hybrid simulator.

In our LSFs, the OS of a language is modeled using the Statechart formalism. Due to the fact that all operational semantics share some concepts (e.g., the current state of the simulator and the operational transitions between states), we have taken the process one step further by creating a canonical form, as a Statechart, for the OS model.

A generic simulator includes a *main simulator loop* in which it evolves its current snapshot through *macro-steps* and *micro-steps*. In our canonical form, a macro-step always increments the simulated-time whereas the micro-steps do not. Hence, in a normal simulation, the execution of each macro-step is interleaved with multiple executions of micro-steps. Each step (macro or micro) consists of `Prepared` and `Processed` phases. The gray states inside the dashed box in Figure 5, and the transitions between them, summarize the generic canonical structure of the OS. The transitions are annotated with descriptions of expected actions, according to our experience.

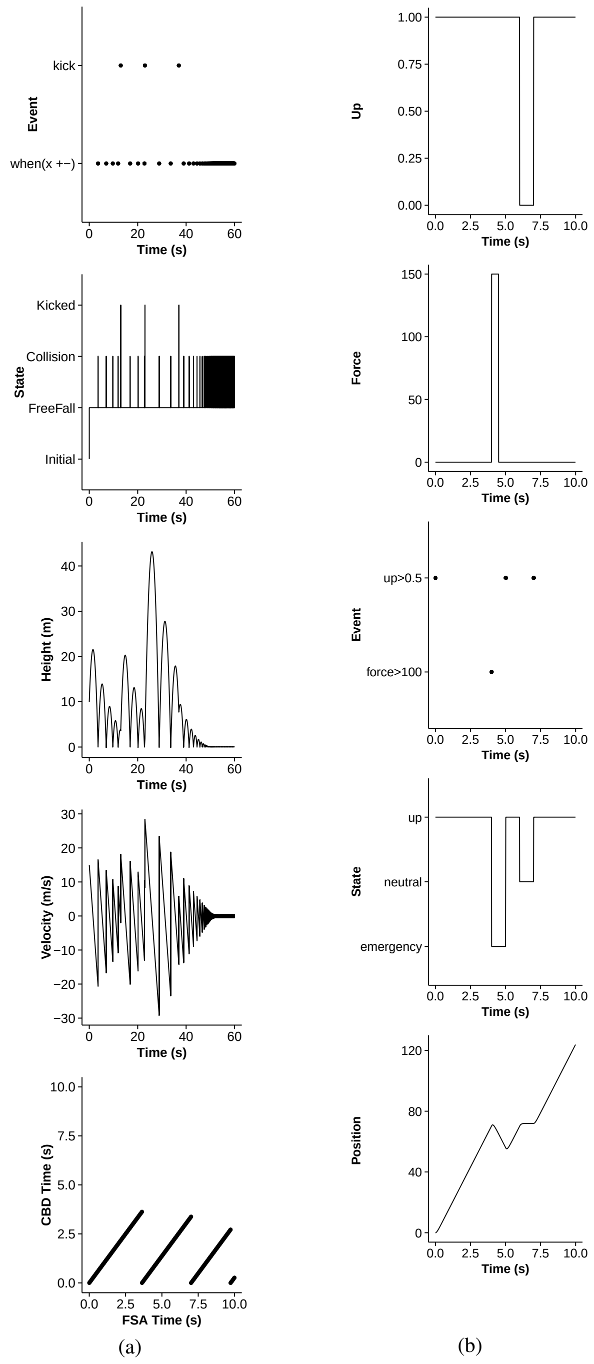


Figure 3. Bouncing Ball (a) and Power Window (b) Results.

It should be noted that it is still up to the language engineer to complete the events, guards, and actions, specific to each formalism. The canonical form enables the identification of *pausing* points where the control can be transferred from one simulator to another (see Section 4).

3.1 TFSA LSF

The OS model describing the TFSA simulator behavior in the canonical form is represented with solid blue states (and

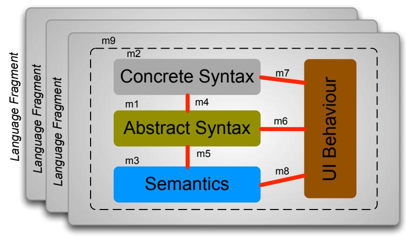


Figure 4. Language Specification Fragments (LSF).

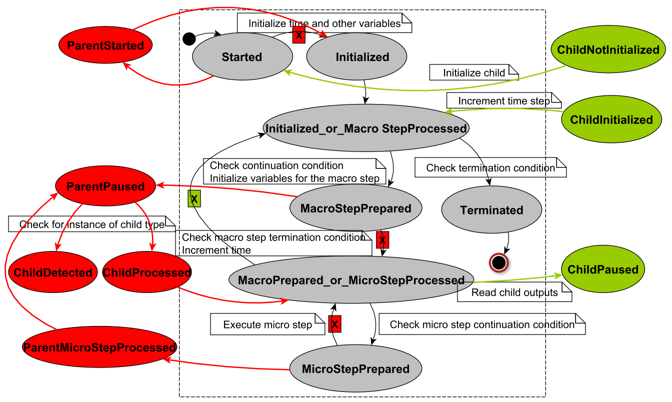

ParentStarted
Initialize time and other variables
X
Started
Initialized
ChildNotInitialized
Initialize child
Increment time step
ChildInitialized
Initialized_or_Macro StepProcessed
Check continuation condition
Initialize variables for the macro step
Check termination condition
ParentPaused
X
MacroStepPrepared
Terminated
Check for instance of child type
Check macro step termination condition
Increment time
X
ChildDetected
ChildProcessed
MacroPrepared_or_MicroStepProcessed
ChildPaused
Read child outputs
Execute micro step
X
Check micro step continuation condition
ParentMicroStepProcessed
MicroStepPrepared

Figure 5. Canonical Form for OS Model.

transitions between them, including the grayed-out ones) on the left of Figure 6. The OS models are not shown separately due to space constraints. A macro-step in the TFSA simulator consists of: 1. reading any event at the current time and checking whether any transition is enabled in the TFSA model (transition P4); 2. taking as many transitions, in the TFSA model, as they become enabled (transitions P8 and P15), one per micro-step; 3. advancing time and finishing the macro-step, if no transition is enabled.

3.2 CBD LSF

The OS model describing the semantics of the CBD simulator is shown on the left in Figure 7. The solid orange states, along with the corresponding transitions (including the grayed-out ones), represent the CBD OS model. For more details on CBD simulation, please refer to [17]. A macro-step consists of: 1. creating a dependency graph and a strong component list (transition P4); 2. iterating and computing every strong component, in order (transitions P15 and P14), one per micro-step; 3. advancing the simulation time after the last strong component is computed, thus finishing the macro-step; The computation of each block depends on its type [1].

4. COMPOSITION

Language composition involves weaving the LSFs of two languages, including syntax and operational semantics (OS). We describe the composition of the abstract syntax in [15]. The composition of the OS involves the composition of time bases, and the interleaving of the control and data flow [4]. Because our scope is restricted to hybrid languages formed by embedding a CTS in a DES and vice-versa, we naturally get a parent-child relationship between the simulators.

To make each simulator a reusable LSF, the OS models are instrumented with composition-specific states used to define interruption points without affecting soundness properties of the simulator execution (see Section 6). Figure 5 shows the instrumentation (extra states and transitions) required at the parent end (in red on the left), and at the child end (in green on the right). To ensure deterministic execution, some transitions – marked with a cross mark – are disabled during the instrumentation process. For example, in the parent, once a macro-step is prepared, it is always paused to allow detection of any embedded model. If no such model is present, the execution continues and the simulator processes the next micro or macro-step. Otherwise, the behavior depends on the protocol for this composition (discussed later in this section). The particular states at which the simulator is paused also depend on the protocol in use. After the instrumentation, the two OS models are woven together, connecting the parent states with the child states – control-flow adaptations – and inserting actions that perform the time and data semantic adaptations.

4.1 Hybrid TFSA

The hybrid TFSA language, introduced in Section 2.1, includes the following extra concepts. **Top-most I/O ports** – the CBD I/O ports of the top-most CBD (the rectangles with embedded triangles in Figure 1) are interpreted as getters/setters of a global storage dictionary, indexed by the name of the port. Pure CBD models, in order to be executable without any adaptations, must not contain any I/O ports at the top-most level. In the hybrid TFSA language, this restriction is relaxed to allow the transmission of values among the CBDs and the parent TFSA. **`When` transitions** become enabled when a zero-crossing is detected in an output port of the source state.

Figure 6 presents a Statechart that models the behavior of the composed language. The disabled transitions of the original Statecharts are grayed-out. The states are color-coded to distinguish between the parent (blue), child (orange), and the adaptation in between (green). The gradient states are added during instrumentation.

To help in understanding how the two simulators interact, we present protocol models for two typical simulation scenarios: 1. the ball is in `FreeFall` state, has not touched the ground, and there are no kick events (Figure 8a); 2. the ball is in `FreeFall` state and touches the ground (Figure 8b). The protocol models, based on the synchronization models presented in [6], raise the level of abstraction of the interactions between the TFSA (DES) and the CBD (CTS) simulators by retaining only the macro and micro-step executions. The snapshots of each simulator are represented as squares in

[1]Our CBD simulator does not support adaptive time-step. This simplifies without affecting our contribution because we are interested in coming up with a woven simulator assuming a fixed set of capabilities from the individual simulators (which excludes the rollback capabilities required to perform adaptive step size).

a 2D space. The coordinates are natural numbers whose purpose is to represent a distance measure between steps. The arrows are numbered in the order of execution and they represent step execution or control transfer. The $6 \approx 1$ arrow states that the protocol should be re-applied if the scenario on which it should be applied still holds. The *last* keyword indicates that there are no micro-steps performed after.

Consider the bouncing ball model presented in Figure 1. For the first simulation scenario, the woven simulator of Figure 6 repeatedly executes transitions P4, P5, P6, H2, C9, C4, C5, C6, C7, C8, H3, H4, P7, P12. Transitions C6 and C7 are repeated as long as there are strong components to be processed in the CBD simulator. In its essence, each macro-step of the TFSA simulator is executed by performing one macro-step of the CBD simulator, including its possible many micro-steps, and then advancing time in the TFSA simulator (protocol of Figure 8a). In the second simulation scenario, the simulator executes transitions P4, P5, P6, H2, C9, C4, C5, C6, C7, C8, H3, H4, P7, P8, P9, P10. The difference from the first scenario is that a `when` transition becomes enabled and the TFSA simulator executes a micro-step (from P7 onward). The protocol in Figure 8b shows only the difference with the previous scenario, to avoid repeating all the steps. Whenever the TFSA simulator executes one micro-step (changing the current state of the model being simulated), the CBD simulator (including its simulation time) is reset and assigned a new model for simulation (see the last plot of Figure 3a). The time reset is clear in the protocol of Figure 8b but not as clear in Figure 6, highlighting the importance of raising the level of abstraction for the step synchronization of the two simulators.

To summarize, the woven simulator incorporates the following semantic adaptations: **Time** – The time-step used in a hybrid TFSA simulation is given by the `getDelta()` function (transition P1), which, in our particular implementation, returns the greatest common divisor (GCD) of the time steps proposed for each model. **Control** – The protocol models summarize the control adaptations. To save space, we omit the initialization scenarios. **Data** – We use a double buffered global dictionary to transmit data from one simulator to the other and detect threshold crossings (transitions C1 and C8).

4.2 Hybrid CBD

The Hybrid CBD language, introduced in Section 2.2, includes the following extra concepts: **Inequality** – A transition can become enabled by checking whether an inequality holds for the current values of the input port signals. **Valued state** – Each state in the embedded TFSA models is associated with one value, that defines the value of the output port of the enveloping block. This is much like including a restricted form of *enter actions* (as defined in Statecharts), such that only a single assignment is allowed to the output port variable.

Figure 7 presents the results of weaving the OS models of each simulator. The woven simulator is color-coded in a similar manner as Figure 6. It is easy to see that the TFSA and the CBD simulated times are kept synchronized at the end of each macro-step of the CBD simulator by observing the protocol model in Figure 8c. The typical simulation follows through transitions P4, P5, P6, P7, H2, C9, C4, C5, C6, C7, C8, P8, and P9. The transitions C6 and C7 might be repeated as long as there are enabled transitions in the TFSA model. In this scenario we assume that the block being processed in the micro-step of the Hybrid CBD is an `FSABlock`. Notice that whenever a new `FSABlock` is going to be computed on the CBD simulator side, the TFSA simulator stores its state and restores a new state associated with that block. This allows multiple `FSABlocks` to be simulated.

To summarize, the woven simulator of Figure 7 is the result of the following semantic adaptations: **Time** – The time-step is given by the GCD of the time-steps proposed for each model. **Control** – The protocol model of Figure 8c summarizes the control adaptations. **Data** – We use a dictionary to hold the complete state of the TFSA simulator associated with each TFSA block (transitions C1, C8 and C9). The detailed code of the simulators is available for download [2].

5. METHODOLOGY

In the previous sections, we described the case studies and their compositions in a bottom-up approach. In this section, we discuss the methodology to build the hybrid languages. To begin with, we follow the technique of de/reconstructing of the simulator as proposed in [20] to build the OS model. The technique entails extracting the modal part of the simulator, and modeling it as a Statechart. This model is then instrumented with extra operations and then re-constructed.

Figure 9 summarizes our process for explicitly modeling the semantic adaptations. The inputs to this process are as follows: **Non-Modal OS** – a library implementing the main operations of the simulator. For the CBD simulator this includes the creation of the strong component list (transition P4 of Figure 7). **Modal OS** – a Statechart modeling the behavior of the simulator, in which the transitions contain actions implemented in the non-modal part. **Synch** – a set of protocols (as shown in Figure 8), describing how the two simulators behave in different simulation scenarios. The protocols are generic in the sense that they make no concrete reference as to which code (and which simulator) implements the macro/micro-steps. **`SemanticMapping`** – maps the macro-step and micro-step abstract concepts in the Synchronization protocols to action code in the transitions of the (modal) OS model.

The **Normalization** process turns the OS model into a canonical form following the guidelines presented in Figure 5. The non-canonical form is exemplified in [15]. The **Instrumentation** process takes the canonical form and adds states/transitions to pause and resume the simulator. The **Composition** process takes the instrumented OS models and weaves them together according to the synchronization protocol. There is enough information in the set of protocols and the semantic mappings to make this composition because the set of protocols describes all possible simulation scenarios. As an example of the weaving, the transitions C8 and P8 in Figure 7 are there because the arrow labeled 4, in Figure 8c,

[2] `http://msdl.cs.mcgill.ca/people/claudio/mle/hybridsimulators.zip`

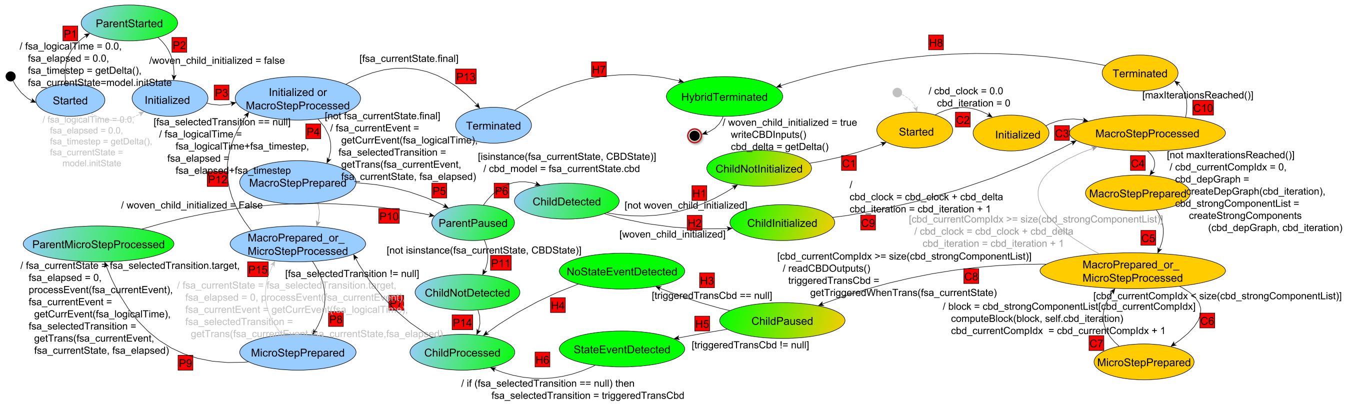


Figure 6. Hybrid TFSA Operational Semantics.

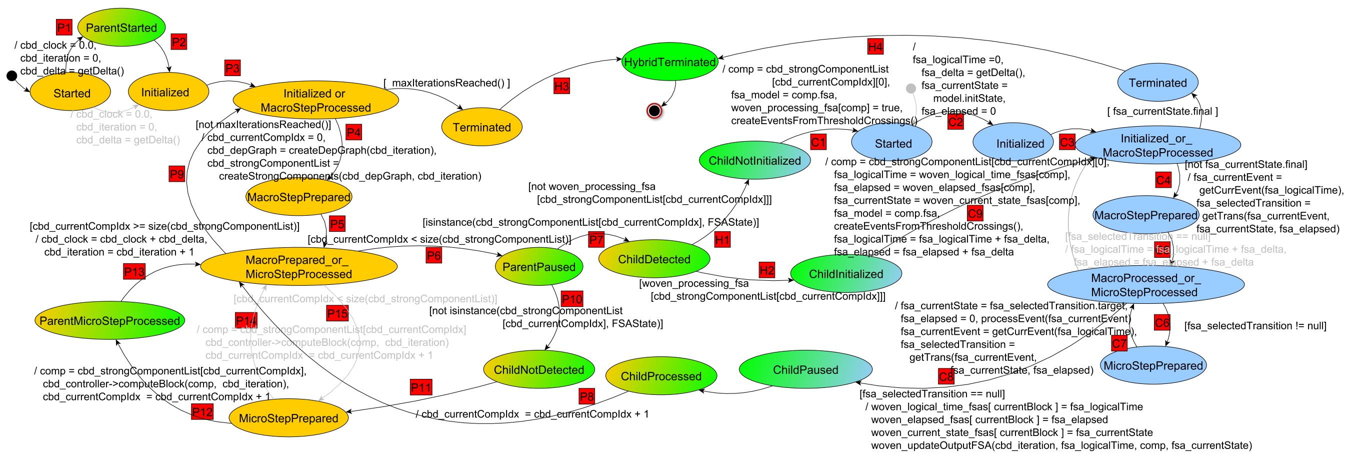


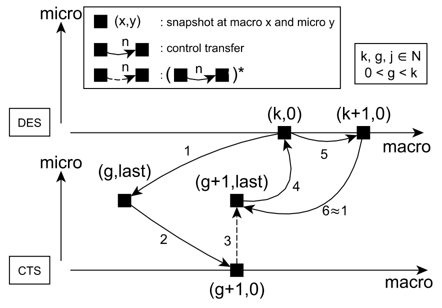
Figure 7. Hybrid CBD Operational Semantics.

performs a micro-step in the CBD simulator. A micro-step in the CBD simulator is mapped (in the semantic mapping model) to the increment of the `cbd_currentCompIdx` variable and computing the current block. Since computing the current block has no effect in a `FSA_Block`, transition P8 only increments the variable. Data and time adaptations are also performed in this process. However, we currently do not have a way to represent those adaptations in a generic way as we did with the control adaptations. However, we intend to represent them as an adaptation mapping model. The **Re-construction** process adds the concrete implementation code to the functions that were added in the composition process [20]. As an example, in our case study we added the `getDelta` function in the composition process. The implementation of this function is the GCD of the deltas proposed for each model.

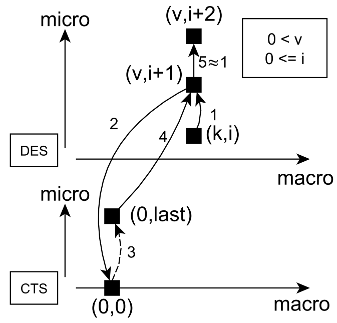
6. DISCUSSION

We propose a generic process for modular hybrid language design re-using simulators from the DES and the CTS domains. Generic because we use a canonical form of representative simulators from both domains. Modular because the synchronization protocol is independent of the simulators. This is especially important since the synchronization protocols are not trivial to build [6] and must ensure correctness properties: **Completeness** – the protocols cover all possible scenarios, i.e., the behavior of the woven simulator is always defined. **Determinism** – the scenarios covered are mutually exclusive, i.e., the behavior of the woven simulator is always unique. **Step progression** – macro-steps always advance. **Step synchronization** – there is an algebraic relationship, not necessarily always the same one, between the macro-steps of the simulators at specific stages. For instance, in the protocol model of Figure 8c, the macro-step k of the CTS simulator is the same as the macro-step of the DES simulator when the final micro-step is executed in the CTS simulator. **Fairness** – at each macro-step, all simulators get to execute. Our aim is that, once proved correct, as was done in [6], the protocols can be reused for other compositions.

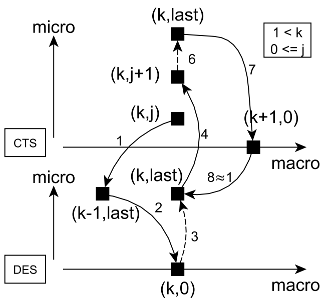
A woven simulator must not only satisfy the same properties as the set of protocol used to create it, but also must ensure *language continuity*, i.e., if the model to be simulated is an instance of just one of the composed languages, then the simulator should still behave as the original simulator of that language. Obviously, in order for the woven simulator to satisfy



(a) DES-CTS – No micro-steps executed.



(b) DES-CTS – Micro-steps executed.



(c) CTS-DES.

Figure 8. Synchronization Models.

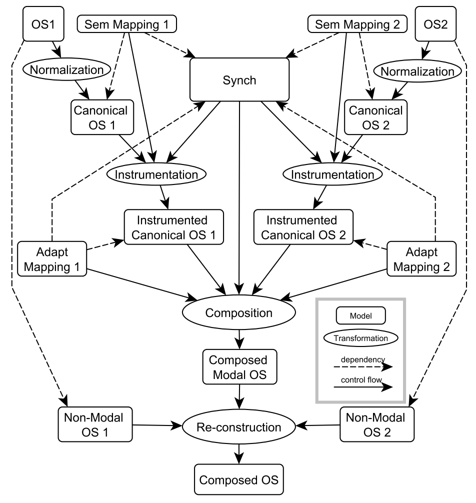


Figure 9. Semantic Adaptation Process.

the correctness properties, the composition process must be correct. This motivates the need for an automatic composition process. However, a few challenges remain to be solved. We intend to implement an automatic process that generates, from the protocols, linear temporal logic constraints to be checked against the woven simulator to ensure its correctness. Currently, the data and time adaptations are performed manually but we intend to create a generic representation for these, based on the language proposed in [12], to be used as input to the automatic composition process. Moreover, we plan on validating our progression by composing other languages such as DEVS [22] and Bond-graphs [16].

Finally, our long-term aim is that the composition process be closed-under-composition, i.e., the resulting woven simulator is a LSF that can be composed again with other languages, to provide arbitrarily complex modeling languages for the development of ever more complex CPS.

7. RELATED WORK

Modularization in MDE, which involves reuse and abstraction of models and model elements expressed in arbitrary domain-specific models, brings about many challenges. The reuse and composition of structural models (or abstract syntax models) have been addressed in FRAGMENTA [2], Spoofax [10], and MPS [21] However, to the best of our knowledge, the reuse of language semantics mostly remains unaddressed in the current state of the art, and is seen as one of the main challenges in modular language engineering. This is largely due to the disparate models of computation employed in the semantics of domain-specific modeling languages. The problem of composing language semantics was initially discussed in [11] and in the works of Ptolemy [7].

Boulanger et al. [3] presents a framework, ModHel'X, for semantic adaptation of heterogeneous models by allowing the user to program an interface block between a main model and an embedded model. The adaptation in ModHel'X is done at the model level, whereas in our work the focus is on the adaptation of the simulators. This means that the composed language is created, and the modeler uses it without any concern about its semantics.

Meyers et al. [12] proposes a domain-specific language (DSL) that allows the user to explicitly model the semantic adaptation blocks. The DSL is used in [5] to generate the semantic adaptations required to enable co-simulation of independent and heterogeneous simulators. Co-simulation is a technique to couple multiple simulators, each simulating a single component, often seen as a black box, in order to perform simulation of the whole system. Our approach does not require that simulator provides an interface to receive and send data. However, we do require that simulators follow a canonical form. Contrasting to the co-simulation domain, where the operational semantics of each language can be considered as a black-box during language composition [4], we follow a white-box approach that allows us to the explicitly model the composition of LSFs by taking into consideration the concepts that rule each of the languages.

8. CONCLUSION

We propose a generic methodology based on language engineering techniques, to develop new modeling languages by composing – and thus reusing – formalisms from both the discrete event and continuous time domains. We focus on the composition of semantics, and address adaptations of data, time, and control. We demonstrated our approach with the use of two case studies: hybrid TFSA and hybrid CBD (compositions of the TFSA and CBD languages).

Our aim is that ever more complex cyber physical systems (CPS) can be modeled with formalisms created on-demand, from the domains that the CPS encompasses. Furthermore, we enable the reuse of one of the most challenging aspects of developing hybrid formalisms – the time synchronization protocol – by introducing a generic protocol language, independent of any concrete simulator, based on the common notion of macro and micro-steps.

9. ACKNOWLEDGMENTS

This work was partly funded by the Automotive Partnership Canada (APC) in the NECSIS project.

REFERENCES

1. Alur, R., and Dill, D. L. A theory of timed automata. *Theoretical Computer Science 126*, 2 (1994), 183–235.
2. Amalio, N., de Lara, J., and Guerra, E. Fragmenta: A theory of fragmentation for MDE. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on* (2015), 106–115.
3. Boulanger, F., and Hardebolle, C. Simulation of Multi-Formalism Models with ModHel'X. In *Software Testing, Verification, and Validation* (2008), 318–327.
4. Boulanger, F., Hardebolle, C., Jacquet, C., and Marcadet, D. Semantic adaptation for models of computation. In *Application of Concurrency to System Design*, IEEE (2011), 153–162.
5. Denil, J., Meyers, B., Denil, J., Meyers, B., Meulenaere, P. D., and Vangheluwe, H. Explicit Semantic Adaptation of Hybrid Formalisms for FMI Co-Simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, Society for Computer Simulation International, Ed. (Alexandria, Virginia, 2015), 99–106.
6. Gheorghe, L., Bouchhima, F., Nicolescu, G., and Boucheneb, H. Semantics for Model-based Validation of Continuous/Discrete Systems. In *Design, Automation and Test in Europe*, DATE '08, ACM (New York, NY, USA, 2008), 498–503.
7. Goderis, A., Brooks, C., Altintas, I., Lee, E. A., and Goble, C. Heterogeneous composition of models of computation. Tech. Rep. UCB/EECS-2007-139, EECS, University of California, Berkeley, 2007.
8. Hardebolle, C., and Boulanger, F. Exploring Multi-Paradigm Modeling Techniques. *Simulation 85*, 11-12 (2009), 688–708.
9. Henzinger, T. The theory of hybrid automata. In *Logic in Computer Science, 1996* (July 1996), 278–292.
10. Kats, L. C., and Visser, E. The spoofax language workbench: Rules for declarative specification of languages and ides. *SIGPLAN Not. 45*, 10 (Oct. 2010), 444–463.
11. Lacoste-Julien, S., Vangheluwe, H., de Lara, J., and Mosterman, P. J. Meta-modelling hybrid formalisms. In *Computer Aided Control Systems Design, 2004 IEEE International Symposium on* (Taipei, 2004), 65–70.
12. Meyers, B., Denil, J., Boulanger, F., Hardebolle, C., Jacquet, C., and Vangheluwe, H. A DSL for explicit semantic adaptation. In *MPM, MoDELS '13* (2013), 47–56.
13. Mosterman, P. J., and Biswas, G. A theory of discontinuities in physical system models. *Journal of the Franklin Institute 335*, 3 (1998), 401–439.
14. Mosterman, P. J., and Vangheluwe, H. Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation 80*, 9 (2004), 433–450.
15. Mustafiz, S., Barroca, B., Gomes, C., and Vangheluwe, H. Towards Modular Language Design using Language Fragments: The Hybrid Systems Case Study. In *Information Technology - New Generations (ITNG), 2016 13th International Conference on* (2016), to appear.
16. Paynter, H. M. *Analysis and design of engineering systems*. MIT press, 1961.
17. Posse, E., de Lara, J., and Vangheluwe, H. Processing causal block diagrams with graphgrammars in atom3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)* (2002), 23–34.
18. Prabhu, S. M., and Mosterman, P. J. Model-based design of a power window system: Modeling, simulation and validation. In *Proceedings of IMAC-XXII: A Conference on Structural Dynamics, Society for Experimental Mechanics, Inc., Dearborn, MI* (2004).
19. Van der Auweraer, H., Anthonis, J., De Bruyne, S., and Leuridan, J. Virtual engineering at work: the challenges for designing mechatronic products. *Engineering with Computers 29*, 3 (2013), 389–408.
20. Vangheluwe, H., Denil, J., Mustafiz, S., Riegelhaupt, D., and Van Mierlo, S. Explicit Modelling of a CBD Experimentation Environment. In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*, DEVS '14, Society for Computer Simulation International (San Diego, CA, USA, 2014).
21. Völter, M., and Visser, E. Language Extension and Composition with Language Workbenches. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, ACM (New York, NY, USA, 2010), 301–304.
22. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*, 2 ed. Academic press, 2000.