Towards Modular Language Design using Language Fragments: The Hybrid Systems Case Study

Sadaf Mustafiz[1], Bruno Barroca[1], Cl'audio Gomes[2], and Hans Vangheluwe[1,2]

[1] McGill University, Montreal, QC, Canada
[2] University of Antwerp, Antwerp, Belgium
{sadaf, bbarroca, hv}@cs.mcgill.ca, claudio.gomes@uantwerpen.be

Abstract. Cyber-physical systems can be best represented using hybrid models that contain specifications of both continuous and discrete event abstractions. The syntax and semantics of such hybrid languages should ideally be defined by reusing the syntax and semantics of each components' formalisms. In language composition, semantic adaptation is needed to ensure correct realization of the concepts that are part of the intricacies of the hybrid language.
In this paper, we present a technique for the composition of heterogeneous languages by explicitly modelling the semantic adaptation between them. Each modelling language is represented as a language specification fragment (LSF): a modular representation of the syntax and semantics. The basis of our technique is to reuse the operational semantics as defined in existing simulators. Our approach is demonstrated by means of a hybrid language composed of timed finite state machines (TFSA) and causal block diagrams (CBD).

1 Introduction

A typical cyber-physical system (CPS) exhibits complex behaviour that can only be best represented using a mix of continuous (i.e., mathematical differential equations) and discrete (logical computation) models. In particular, to engineer such systems, we need to first study how existing physical laws and quantities, co-exist and interact with logical controllers, which themselves are also bound to the same laws. The engineering of a CPS involves the modelling and simulation of hybrid models (e.g., combinations of sets of piece-wise continuous functions alternating with events). Moreover, due to the extreme complexity of the CPS, it is often the case that we need to break the models into several orthogonal views, aspects or quantities (e.g., thermal, electric, power), in order to be able to study the behaviour of each of them over time, and only afterwards study their integration and interaction.

It is therefore obvious that the success of building such complex CPS can only be maximized if one first identifies the most appropriate level of abstraction to break the complexity in each of these orthogonal views, and then uses the most appropriate formalisms that somehow realize these abstractions, while

bringing to the CPS engineers a valuable engineering toolbox that integrates several CPS modelling and simulation environments. However, we advocate that the CPS engineer should (instead of being yet another language engineer) become a language integrator, by reusing and integrating existing languages into a new language that fits the expressiveness needs of a particular CPS. Unfortunately, the existing techniques for language modularity are still not adequate to fully realize this objective. From our recent work [12], the need for techniques to modularize the design of modelling languages has become apparent to us.

In this paper, we underline new techniques for language modularity based on the concept of Language Specification Fragments (LSF). Such fragments include the syntax (both abstract and concrete), the semantics, and the user-interface behaviour, along with the various interleaving of the elements involved. The LSFs form the basis of reuse and abstraction in language design, and allows the development of new formalisms by re-using and merging existing fragments, leading to reduced efforts on the language designers part. For example, an existing platform-dependent DEVS formalism can be extended by integrating an existing Neutral Action Language in it to compose a neutral DEVS formalism [2]. Such compositions become even more useful when integrating UML formalisms, e.g. Statechart and Class Diagrams with Action Code.

Following the initial ideas of meta-modeling hybrid formalisms [10], we first observe and meta-model a particular hybrid formalism that combines a language for Timed Finite State Automata (TFSA) with a language for Causal Block Diagrams (CBD). We then perform a conceptual de-construction of the hybrid formalism into two (reusable) LSFs, while identifying the composition operations that are required in order to re-construct it back to the original formalism. Contrasting to the co-simulation domain, where the operational semantics of each language is considered as a black-box during language composition (otherwise referred to as semantic adaptation) [7], 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.

This paper is structured as follows: Section 2 gives an introduction to the case study used, Section 3 introduces language specification fragments, and the LSFs used in our case study. Section 4 describes our language composition technique by means of the case study, and Section 5 discusses possible means to generalize and automate the fragmentization and composition process. Finally, Section 6 presents related work and Section 7 concludes with future work.

2 Hybrid Language Case Study

We introduce here the case study, the Hybrid TFSA-CBD language, that is used throughout this paper to demonstrate the composition of language fragments.

2.1 The Hybrid TFSA-CBD Language

The hybrid case study is essentially the hybrid language case that can effectively model hybrid systems, i.e., systems that exhibit both discrete and continuous

behaviour in the form of piece-wise continuous interleaved with discrete events. We have taken a basic language from each of the domains, TFSA (discrete) and CBD (continuous), to study the interleaving of the behaviour of the two languages and the two time domains.

Timed Finite State Automata (TFSA) are used for describing behaviour of reactive systems. A TFSA consists of the following: a set of states including a start stat ; a set of transitions between the defined states, that include triggers/events (and/or guards) that can be either an event name or an `after` indicating some delay in time; and finally, a set of input events tagged in time.

Causal Block Diagrams (CBDs) is a visual modelling 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. Each block has (optional) inputs and one output[3], and it can either represent an algebraic mathematical operation (such as summation, multiplication) or a time sensitive operation (e.g., delay the input). In our work, we focus on continuous-time models represented as CBDs. The simulation of such CBDs are however usually carried out in digital computers using a discrete-time approximation (i.e., by discretizing the differential equations and translating them into difference equations, which are represented as discrete-time CBDs) [6].

A TFSA-CBD composition is the weaving of a TFSA and CBD together in possibly different ways: 1) TFSA composed of a CBD; 2) CBD composed of a TFSA; or 3) a hierarchical composition with a TFSA composed of a CBD which in turn is composed of a TFSA (TFSA-CBD-TFSA or even CBD-TFSA-CBD). In this paper, our case study specifically focuses on the first kind of composition, that is a CBD embedded within a TFSA state where TFSA is the parent language and CBD is the child language. This is similar in concept to that of having Stateflow® models within Simulink® models.

When simulating dynamical systems modelled as a CBD, the output of the difference equations (described in the CBD) becomes not only a function of the input, but also a function of the whole history of inputs and the initial conditions, due to the use of time sensitive blocks (e.g., delay). The history of inputs is stored in the form of state variables and so the output is calculated as a function of the inputs and the state. This not only implies that the detection of enabled transitions must be done at the end of each simulation step of the CBDs, but also means that a mechanism must be in place to ensure that after a transition occurs to a state with a CBD contained, proper initial conditions are provided to that CBD [13]. The transitions are usually triggered via events or guards that are usually implemented using *if* statements. In our case however, we want to know *when* the transition occurs as opposed to *if* the transition has occurred. For this purpose, *monitoring functions* need to be defined that allow for zero-crossing detection. The detection of the exact zero-crossing time is non-trivial and requires the use of state-event location techniques involving rollbacks

[3] Without loss of generality, we assume here that blocks in a CBD only have one output.

to find the exact point. The syntax and semantics of the hybrid language needs to address this requirement. Additionally, modellers should also be given support to create hybrid models with the appropriate syntax to model such situations.

2.2 A Hybrid TFSA-CBD Example

The bouncing ball is a classic example of a hybrid system displaying both continuous and discrete behaviour. A ball is in free-fall motion when dropped. The dynamics change when the ball collides with the ground and bounces up again with reduced energy. An external event, such as a kick, is represented here, for illustrative purposes, as a constant increment in the velocity of the ball.

Fig. 1 shows the hybrid model of a bouncing ball modelled using our composed TFSA-CBD language. It includes the *freefall*, *collision*, and *kicked* states with state changes triggered by discrete events. Within each state, the continuous dynamics of the ball (essentially modelled with its height, x, and the velocity, v) evolves with time. 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 (detected by the `when +-` event), and the new v_0 is given by $v_0 = -0.8v$. In the same instance, the ball goes back to free falling again.

3 Hybrid TFSA-CBD Language Fragments

3.1 Language Specification Fragments

Language fragments can be defined as reusable components in the language engineering process. The fragments allow both reuse and modular language design. We define a language specification fragment (LSF) as a description of the specification of a modelling language in terms of its abstract syntax (AS) , concrete syntax (CS), operational semantics (OS), and the user-interface (UI) behaviour. As presented in Fig. 2, such a specification involves the definition of several models (referred to as m1-m9).

- m1: The AS model (at times referred to as the linguistic type model) describes the essence of the language (its structure and elements) by means of a meta-model usually modelled as a class diagram or an entity-relationship model.
- m2: The model instances of the language needs to be defined using some CS: a visual syntax, a textual syntax, or a combination of both. A single AS may be associated with one or more CS models.
- m3: The semantic model describes the language OS. In the case of a modelling and simulation environment, this refers to the simulator semantics defined with an algorithm outlining the computation steps.
- m4: The mapping model defining the mapping and rendering links between the AS and one CS.
- m5: The semantic mapping model which specifies how the AS elements are treated in the OS.
- m6-m8: These models specify the mappings between the UI behaviour of the interactive modelling environment and the AS (m6), CS (m7), and OS (m8).
- m9: A *glue model* that binds together models m1-m8.

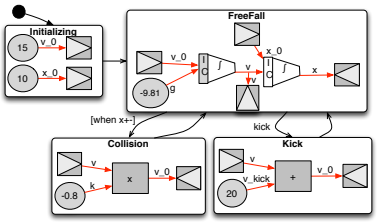

Fig. 1: Bouncing Ball Hybrid TFSA-CBD Model

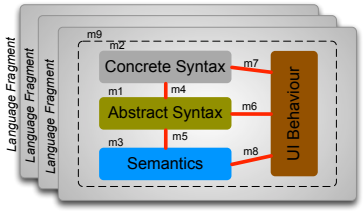

Fig. 2: Language Specification Fragments (LSF)

This set of models together define one LSF, which in turn describes a part of the overall semantics of a hybrid language. Our focus here is on the AS and the semantics (models m1, m3, and m5), and we intend to look at the interfaces and a composition mechanism to compose the AS as well as the OS.

3.2 TFSA and CBD Fragments

We present here the TFSA and CBD LSFs. Fig. 3 presents the AS model of the TFSA language as a class diagram. Algorithm 1 outlines the behaviour for the FSA simulator. Besides the traditional notions of states and transitions, where on each transition we have the event triggers, it also includes the `after` construct that is implemented with the implicit notion of *elapsed time*.

Fig. 4 presents the AS model of the CBD language as a class diagram. Algorithm 2 outlines the behaviour of the CBD simulator in the *main loop* of the algorithm. For now, we are assuming fixed time-step simulation. This simplifies the zero-crossing detection without compromising the strength of our contribution because we are interested in coming up with the possible weavings assuming a fixed set of capabilities in the simulators.

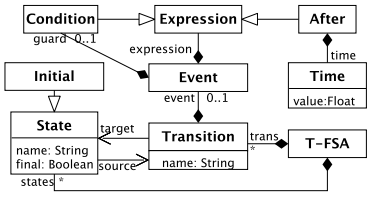

Fig. 3: TFSA Abstract Syntax Model

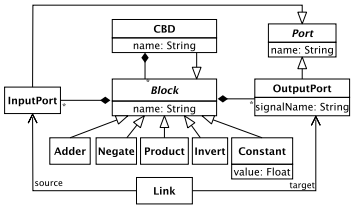

Fig. 4: CBD Abstract Syntax Model

4 Hybrid TFSA-CBD Language Composition

In this section, we describe the composition of the hybrid TFSA-CBD language with regards to the abstract syntax and the semantics. In our case study, the hybrid system requirement is to model the continuous dynamics of a system when the system is in certain states. This entails TFSA states to be *embedded* with CBDs. Hence, a single state in a TFSA can be a simple state or a CBD.

Weaving the language descriptions defined as fragments as per our requirements gives us a new language definition (which can then again be referred to as a

Algorithm 1 TFSA Operational Semantics

```
logicalTime, elapsedTime ← 0; currentState ← initialState
while not endCondition do
    E ← getInputEventAt(logicalTime)
    if out-transition T from currentState has E then
        currentState ← currentState.T.destination;
        removeInputEventFromInputList(); elapsedTime ← 0
    end if
    if out-transition T from currentState has after(time) & time ≤ elapsedTime then
        currentState ← currentState.T.destination; elapsedTime ← 0
    end if                                          ▷ Δt is a parameter to this algorithm
    logicalTime ← logicalTime + Δt; elapsedTime ← elapsedTime + Δt
end while
```

Algorithm 2 CBD Operational Semantics (Adapted from [17])

```
logicalTime ← 0
while not end_condition do
    schedule ← LOOPDETECT(DEPGRAPH(cbd))
    for gblock in schedule do
        COMPUTE(gblock)
    end for
    logicalTime ← logicalTime + Δt                  ▷ Δt is a parameter to this algorithm
end while
```

re-usable fragment) with a new AS model (class diagram) and a new automaton with new action code and a bigger store.

4.1 Composition of the Abstract Syntax Models

The AS models (or meta-models) modelled as class diagrams (shown in Fig. 3 and Fig. 4) are composed, resulting in the AS model of the hybrid language. The composition process involves the use of rule-based graph transformations that matches the pre-defined parent and child classes and joins them in a containment relationship within the class diagram. Due to space reasons, Fig. 5 only presents part of the composed class diagram - the complete details of the `Block` class and `Expression` class are shown in the original class diagrams. The `Transition` in the hybrid AS model is adapted to include a special kind of guard, specified (in the concrete syntax) as `[when +-]` to define an event for the zero-crossing detection in the source state dynamics. The *when* condition (both `[when +-]` and `[when -+]`) is added as a specialization of `Expression`.

Composing the AS models along with the CS models in our meta-modelling and model transformation tool, AToMPM (A Tool for Multi-Paradigm Modelling) [16], allows us to generate a working visual editor for the hybrid language. The AS data structures are also composed at the textual model level using SCCDs (described in the next subsection).

4.2 Composition of the Operational Semantics Models

Semantic adaptation involves the composition of time bases, and the interleaving of the control flow and data flow [7]. Fig. 6 shows how the semantics of the two languages, TFSA and CBD, are joined and adapted using operations to allow

the two simulators to work in conjunction. The semantics behind the transitions labelled 1 to 6 are discussed below.

– **Time adaptation**: This entails computing the new hybrid time step based on the outcome of a maximal common divisor (MCD) of the original time steps (see label 1). The CBD clock is initialized to zero at every simulation step when processing control is handed over to the CBD from the FSA (see label 5). The TFSA-CBD clock is incremented when all enabled transitions are processed and the FSA is ready to advance to the next state (see label 4).
– **Control adaptation**: The parent language, TFSA, has initial control. When a CBD is detected inside a FSA state, control is passed on to the CBD (see label 2). Following the execution of a CBD step, the simulator checks for zero-crossings and for any external events in the TFSA (see label 3). If a zero-crossing is detected, the CBD is reinitialized and execution continues (see label 2). An external event being triggered might involve a change in state and possibly the execution of a new embedded CBD. It should be noted that the switching of control is rather conceptual in our case, since our hybrid simulator is one combined simulator and does not require the passing of control from one simulator to another.
– **Data adaptation**: The global state of the model is updated at every iteration at the child level and at the parent level (see labels 2 and 3).

The OS in algorithm form (presented in Section 3) is mapped to an automaton (SCCD model) with details in action code for each fragment involved. SCCD is a formalism that combines Class diagrams and Statecharts to define classes, their behaviour and interactions. The weaving of the simulation algorithm automatons (SCCDs) is based on a syntactic and semantic *glue* model, similar to the model shown in Fig. 6. The use of the SCCD language allows us to generate SCCD XML or SCCD HUTN (human-usable textual notation) from a complete behavioural model of a simulator instead of hard coding the simulator and carrying out the adaptation within wrappers at the code level.

The SCCD models describing the OS of the TFSA and the CBD languages are shown as part of Fig. 7, which presents the composed SCCD model for hybrid TFSA-CBD. In the figure, the TFSA and CBD composite states represent the SCCD models of the TFSA and CBD simulators respectively. For space reasons, the internal details of the simulator classes are not presented. These classes include the definition of the data structures, runtime variables, and methods used in the Statechart part of the SCCD model. The visual SCCD is first transformed to a SCCD HUTN model which is then compiled to Python source. This results in a fully automatically generated simulator from our description of the OS.

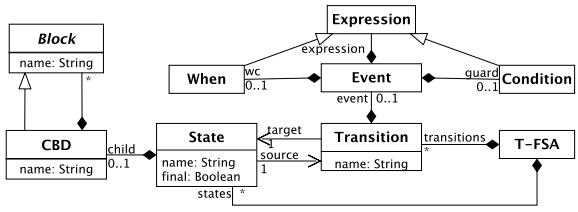

Fig. 5: TFSA-CBD Composed Abstract Syntax Model (Partial Model)

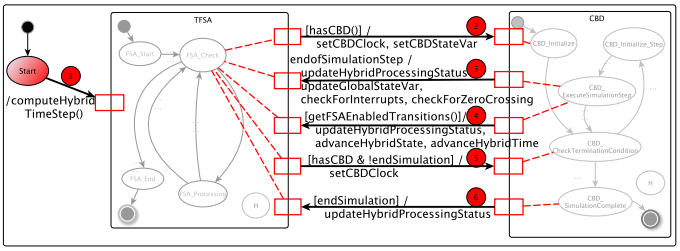

Fig. 6: Hybrid TFSA-CBD Semantic Adaptation

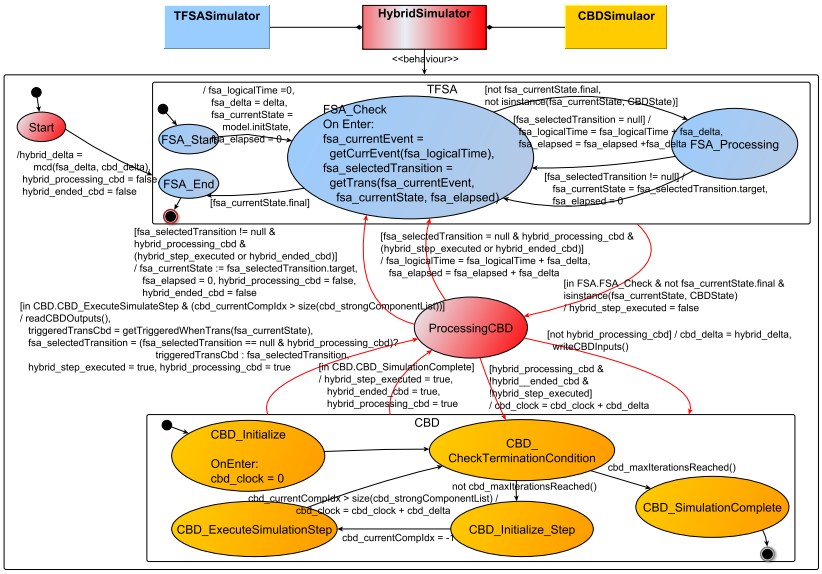

Fig. 7: Hybrid TFSA-CBD Operational Semantics

4.3 Simulation using the Composed Simulator

The bouncing ball model (shown in Fig. 1) was simulated using our woven simulator. Figure 8 presents five graphs all plotted against the *time* of the parent formalism, TFSA in this case. The simulation results shows the zeno behaviour of the bouncing ball. It also shows how the nested child (CBD) clock advances with the hybrid (parent) clock.

5 Modular Language Design of Hybrid Languages

Based on the insights gained from the hybrid case study, we are looking more deeply into the fragmentization and composition process. Ongoing work is discussed here.

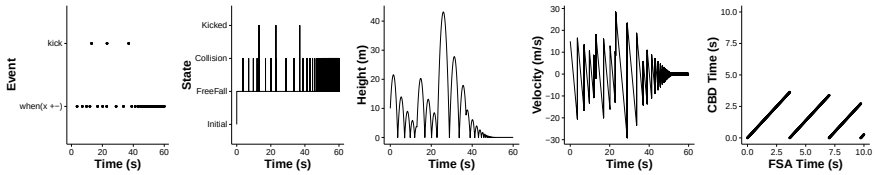

Fig. 8: Bouncing Ball Hybrid Model Simulation Results

5.1 Fragmentization

We propose using a network language (with input/output ports) as a way to package each LSF to enable interleavings with other LSFs via pre-defined ports (similar to what is shown in Fig. 6). The network language provides an interface for each fragment SCCD which can be used to define explicit communication between the LSFs involved and to carry out semantic adaptation. This model can then be automatically transformed to the woven SCCD for the new language. These LSFs ultimately can be used to build a library of reusable fragments.

In this paper, we have defined LSFs to be complete and meaningful languages. This constraint can be further relaxed to allow LSFs to be underspecified or to include *holes*. These model elements or *holes* can then be referenced and linked with special ports in the network language, which can then be replaced or extended with another complex model (defined as a LSF) in any part of the AS or OS model. For instance, a statement can be replaced by a block of statements or an association replaced by an entire meta-model. The nested LSF can itself have *holes* leading to a composed language that can further be replaced. These *holes* must be identified and replaced in order to build valid and meaningful languages. At the moment, we are looking into techniques to allow automatic checking for inconsistencies and incompleteness in LSFs.

5.2 Composition

Based on the studies carried out, we propose five patterns for language composition: *embed*, *weave*, *build*, *replace*, *extend*, and *slice*. Fig. 9 gives an example of each pattern. In the hybrid TFSA-CBD case study, we implicitly used the *embed* pattern to compose the two languages. Embedding leads to a parent-child structure which needs to be addressed in a specific manner in the weaving. The addition of the `when` construct was implicitly done using the *extend* pattern. Having a model transformation defined for each pattern would allow modellers to set the initial parameters for the composition and select the required pattern to build a composed syntax or semantics model. A higher-order transformation could identify the *holes* in the language(s) and define the order in which each pattern should be applied, leading to an automated weaving process. AToMPM will be used to build and compose the AS models using graph transformations.

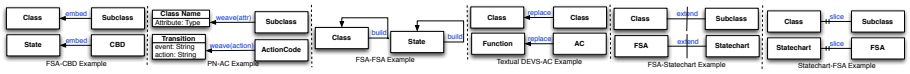

Fig. 9: Language Composition Patterns

5.3 Automating the Semantic Adaption Process

As part of the fragmentization and composition process, the definition and composition of the SCCD models need to be developed further. We begin by generalizing the OS specification by defining a fixed structure that every simulator design has to adhere to. We propose adapting the simulator design to the form of a Statechart (as part of the SCCD) where every simulator has the following (basic or composite) states: *Start*, *Prepare*, *Process*, *CheckTermination*, *EndSimulation*, and a history state. The execution follows in sequence with a loop that returns to *Prepare* following *CheckTermination* to continue with the remaining iterations. To make the simulator a reusable LSF, we next instrument the Statechart with a *ProcessChild* (which now acts as a *hole*) state with transitions to and from *Process*. The *ProcessChild* is then connected to a network port which allows it to be referenced by other LSFs. Once we have a generic structure for all simulators, we can easily use graph transformations to compose the simulators. The semantic adaptation can be carried out by defining interleavings and adding special constructs with the aid of the patterns discussed above.

We envision having a composition process that is closed-under-composition: the woven simulator is a LSF that can be composed again with other languages, to provide arbitrarily complex modelling languages for the development of ever more complex CPS.

6 Related Work

In modular language engineering, the disparate models of computation employed in the semantics of domain-specific modelling languages bring about many challenges in what matters to their reuse. Reuse is the main focus of CORE [15], which allows specification of reusable aspects as *concerns* and provides support for automated weaving of requirements models. In [1], structural models expressed as instances of UML class diagrams, can be seen as graphs and therefore reused across languages. The reuse mechanism is expressed and ruled by the notion of fragments, which are reusable and composable as new fragments.

Some existing approaches focus on reusing models of modelling languages to build new modelling languages, and on trying to identify the most convenient abstractions to convey the reuse of such kinds of models of languages. In [4], several composition operations such as merge, aggregation, and deletion are introduced to allow new languages to be built from more simpler ones. The concepts for language reuse were further developed and studied in language workbenches, such as Spoofax [8] and MPS [18]. Moreover, the problem of language composition from other language components was extensively explored in [9].

In [19], the structure of languages (i.e., abstract syntax and semantics) is taken into account as different roles, in the composition of languages here considered as language components that can be reused. Similar to [19] and [1], the work in [20] introduces a language for the definition of language fragments, which treats languages as components that can be reused and composed.

The reuse of the 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. If we approach this problem by considering only the semantics that are given by translations, then it might be solved using parameterization [14]. This solution though only glosses over the problem of reusing the semantics of languages, and the meaning and behaviour of the translated models in the target language with regards to composability and compatibility are not addressed. However, the work in [5] does tackle semantics composition by means of aspect-oriented concepts, but can be considered to be a code-level solution. A framework for composing semantics for the purpose of simulating multi-formalism models is proposed in ModHelX [3], which involves an explicit representation of the model of computation used in each of the languages. In [11][7], hybrid formalisms are created as the result of composing OS by assuming a common interface abided by black-boxed simulators. The interface ensures that they can be composed while preserving important semantic properties such as a common time-base thereby ensuring that the composition of the language semantics is meaningful.

7 Conclusion

This paper presents a language composition technique for hybrid systems, demonstrated using the hybrid TFSA-CBD case study. We introduce language specification fragments (LSF) to model the specification of languages as a set of syntax and semantics models. The composition technique focuses on fragmentization and composition of the LSFs (specifically, the AS and the OS). The semantics (i.e., the behaviour of the simulators) are explicitly modelled as SCCDs (Statechart+Class Diagram), which are then woven together by applying semantic adaptations which address the problems that arises in hybrid languages (such as, the need for zero-crossing detection). From the composed semantics model, we are able to generate a fully functional hybrid simulator for the TFSA-CBD language. We plan on working on several case studies involving compositions of discrete-discrete and continuous-continuous formalisms, in addition to other discrete-continuous formalisms to further validate our claims.

As future work, we also intend to adapt the composition technique to include the interleaving of the concrete syntax (along with the UI behaviour) in the LSFs.

8 Acknowledgments

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

References

1. N. Amalio, J. de Lara, and E. Guerra. FRAGMENTA: A theory of fragmentation for MDE. In *MODELS*, pages 106–115. IEEE, 2015.

2. B. Barroca, S. Mustafiz, S. V. Mierlo, and H. Vangheluwe. Integrating a neutral action language in a DEVS modelling environment. In *SIMUTOOLS*, pages 19–28. ACM, 2015.
3. F. Boulanger, C. Hardebolle, C. Jacquet, and D. Marcadet. Semantic adaptation for models of computations. In *Application of Concurrency to System Design*, pages 153–162. IEEE, 2011.
4. J. de Lara, E. Guerra, and J. Snchez-Cuadrado. Abstracting modelling languages: A reutilization approach. In *Advanced Information Systems Engineering*, volume 7328 of *LNCS*, pages 127–143. Springer, 2012.
5. T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel. Melange: A meta-language for modular and reusable development of DSLs. In *SLE '15*, pages 25–36. ACM, 2015.
6. B. Denckla and P. Mosterman. Formalizing causal block diagrams for modeling a class of hybrid dynamic systems. In *CDC-ECC '05*, pages 4193–4198, 2005.
7. J. Denil, B. Meyers, P. De Meuleneare, and H. Vangheluwe. Explicit semantic adaptation of hybrid formalisms for FMI co-simulation. In *TMS/DEVS '15*, pages 852–859. SCS International, 2015.
8. L. C. Kats and E. Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45(10):444–463, Oct. 2010.
9. H. Krahn, B. Rumpe, and S. Vlkel. Monticore: a framework for compositional development of domain specific languages. *Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010.
10. S. Lacoste-Julien, H. Vangheluwe, J. D. Lara, and P. J. Mosterman. Meta-modelling hybrid formalisms. In *ComputerAided Control System Design*, pages 65–70. IEEE, 2004.
11. B. Meyers, J. Denil, F. Boulanger, C. Hardebolle, C. Jacquet, and H. Vangheluwe. A DSL for explicit semantic adaptation. In *7th Workshop on Multi-Paradigm Modelling, MoDELS '13*, pages 47–56, 2013.
12. S. V. Mierlo, B. Barroca, H. Vangheluwe, E. Syriani, and T. Kühne. Multi-level modelling in the modelverse. In *Workshop on Multi-Level Modelling, MoDELS*, pages 83–92, 2014.
13. P. J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *Hybrid Systems: Computation and Control*, volume 1569 of *LNCS*, pages 165–177. Springer, 1999.
14. L. Pedro, V. Amaral, and D. Buchs. Foundations for a Domain Specific Modeling Language Prototyping Environment: A compositional approach. In *8th OOPSLA Workshop on Domain-Specific Modeling (DSM)*, Oct. 2008.
15. M. Schöttle, O. Alam, A. Ayed, and J. Kienzle. Concern-oriented software design with TouchRAM. In *MODELS '13*, pages 51–55, 2013.
16. E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. AToMPM: A web-based modeling environment. In *MODELS'13 Demonstrations*, 2013.
17. H. Vangheluwe, J. Denil, S. Mustafiz, D. Riegelhaupt, and S. Van Mierlo. Explicit modelling of a CBD experimentation environment. In *TMS/DEVS '14*, pages 13:1–13:8. SCS International, 2014.
18. M. Völter and E. Visser. Language extension and composition with language workbenches. In *OOPSLA '10 Companion*, pages 301–304. ACM, 2010.
19. C. Wende, N. Thieme, and S. Zschaler. A role-based approach towards modular language engineering. In *SLE '10*, pages 254–273. Springer, 2010.
20. S. Živković and D. Karagiannis. Towards metamodelling-in-the-large: Interface-based composition for modular metamodel development. In *Enterprise, Business-Process and Information Systems Modeling*, pages 413–428. Springer, 2015.