Multi-Paradigm Semantics for Simulating SysML Models using SystemC-AMS

Daniel Chaves Café [*†], Frédéric Boulanger[*], Filipe Vinci dos Santos[†], Christophe Jacquet[*], Cécile Hardebolle[*]
[*]Supelec E3S – Computer Science Departement
[†]Thales Chair on Advanced Analog System Design
Supélec, Gif-sur-Yvette, France
Email: {firstname.lastname}@supelec.fr

***Abstract*—SysML is an industrial standard for the modeling of systems, providing a graphical way to model structure and behavior. Despite its flexibility, SysML lacks semantics to give language elements a precise meaning. Current implementations of the standard allow multiple interpretations of syntactical elements and can cause misunderstandings when porting a model among tools. Our work focuses on the definition of concrete semantics for SysML to enable correct interpretation of heterogeneous models. We also add semantic adaptation elements to guarantee that interactions among different formalisms are unambiguous. We demonstrate our approach by generating SystemC-AMS code automatically from SysML diagrams for a case study with two distinct formalisms. This kind of translation allows the validation of system behavior through simulation.**

***Index Terms*—Model Transformation, Semantic Adaptation, System Modeling, System Simulation, SysML, SystemC-AMS.**

I. Introduction

For years we have managed the complexity of large systems by a component-based approach, breaking down the system in a set of objects composed hierarchically with interactions restricted to tightly defined interfaces. Heterogeneous systems require extra modeling effort because the interactions must be well defined when crossing the boundaries of different domains. Even if the interfaces are completely specified, a problem remains: the execution semantics of components of different domains may differ and without a formal definition of how these components should interact, it is most likely that the overall system, once integrated, behaves unpredictably or in an implementation-dependent way.

During the design phase of a system, it is rather common to use simulations of models as a support to validate correct functionality of a component before deploying it to fabrication. Those models can be either described in textual or graphical languages. The latter is generaly accepted to be easier to use and more comprehensible.

SysML is now an industry standard for a graphical system-level specification language. It is used to manage complexity while improving communication among different teams. The use of SysML diagrams facilitates the documentation and specification regarding system requirements and constraints on property values [1].

SysML does not provide built-in simulation capabilities but offers great flexibility for modeling large heterogeneous systems. For our purposes, heterogeneous systems are systems that present components of different nature which are modeled using distinct formalisms. A heterogeneous model thus involves at least two different modeling formalisms with bound interfaces, i.e. exchanging data, control or sharing the same time scale. The challenge that we face is to give SysML diagrams explicit and executable semantics so that heterogeneous models can have coherent behavior for simulations across different tools.

The meaning of SysML structural elements is not given by the standard itself. Users of SysML commonly give their own (often implicit) interpretations of how each language element behaves. Block definition diagrams and internal block diagrams are examples of purely structural diagrams with no behavioral semantics. A SysML connector, for example, can represent a wire, a physical chain or even a function call. The problem of the lack of clear semantics is more severe in heterogeneous models. The interaction of a finite state machine monitoring inputs from a continuous time model is a classical example. From the SysML perspective, there is nothing that defines explicitly when a guard should be evaluated or with what precision continuous time data should be monitored.

One way to describe the behavior of a model is by defining its Model of Computation (MoC) [2]. The MoC details how components of a given system interact, how they exchange data, control and notions of time. Examples of commonly used MoCs are: Discrete Event (DE), Continuous Time (CT) and Finite State Machine (FSM). These and have been detailed in [3]. The challenge is how to combine them seamlessly so as to run simulations of the whole system with predictable results.

The present work tackles the problem of simulating heterogeneous systems described in SysML by adding concrete semantics to SysML diagrams throught the use of MoCs. We define explicit rules that we call *semantic adaptations* for the interactions between these MoCs and we automatically generate SystemC-AMS code using model transformations based on our semantics definitions. We thereby take advantage of the SysML language as our front-end modeling environment and simultaneously benefit from SystemC-AMS powerful simulation engine to validate heterogeneous systems by simulation.

This article is organized as follows. Section II introduces previous approaches and the state of the art concerning heterogeneous modeling languages, the use of multiple MoCs with

SystemC and the generation of SystemC code from SysML and UML diagrams. Section III introduces our approach and show implementation details of our model transformations. Section IV details the semantics of the two MoCs explored in the example shown in Section V. Then sections VI discusses the advantages and drawbacks of our approach and section VII concludes.

II. Related Work

A. Heterogeneous modeling languages

Modeling heterogeneous systems is not an easy task. The obstacles and issues arising when trying to implement such models have been the focus of considerable research. Ptolemy II [4] handles heterogeneity by hierarchy. Components are nested in black boxes (also called Actors) for which the semantics of execution and communication are defined by an entity called *Director*. A director defines how a model should behave and how its components communicate, in other words, it defines the model of computation (MoC) Actors can be transparent or opaque regarding its parent: if the child actor does not have its own director it is considered to be transparent and will inherit its parent director, if it has its own director, it is considered to be opaque (like a black box). In Ptolemy, computation and communication semantics are well defined for a large set of MoCs. Unfortunately, there is no explicit way to define the adaptation between models that use different MoCs. This can cause some confusion if the user is not aware of the default adaptation performed by Ptolemy. Extra modeling effort may be required if a specific behavior is expected.

Ptolemy's focus on heterogeneous systems inspired other works such as ModHel'X [5]. ModHel'X proposes a flexible framework for the development of heterogeneous systems separating the model's definition from the MoC's definition. In ModHel'X, a generic execution environment was created to allow the definition of several models of computation. Following the same principle of actor-based modeling, ModHel'X defines blocks whose behavior is determined by a MoC (equivalent to Ptolemy's *Director*) and introduces an interface entity capable of making the necessary adaptations among different MoCs (i.e. data, control and time). To do so, ModHel'X improves upon the execution algorithm of Ptolemy by the introduction of an adaptation phase right before and after the "fire" phase. This yields an effective way to define the semantics of the interactions between different models of computation.

B. Multiple MoCs with SystemC

SystemC [6] has clearly established itself as an important system-level specification language providing simulation capabilities in an early phase of development. One important application of SystemC is the description and automatic synthesis of digital hardware [7] from a subset of the language. SystemC is based on a dedicated discrete event (DE) simulation kernel capable of modeling concurrent processes. However, using only a DE MoC can be cumbersome if one needs to model several application domains. Several researchers have extended the DE kernel with libraries in order to implement different MoCs. This strategy often results in performance degradation, as discussed in [8]. H.D. Patel and S.K. Shukla have shown that better implementations could be achieved if dedicated simulation kernels are used for MoCs such as FSM, Communication Sequential Processes (CSP) and Synchronous DataFlow (SDF).

In any case, dedicated simulation kernels do not solve entirely the problem of the heterogeneity. The interactions among different MoCs are hard to model and may result in semantic conflicts. One interesting solution is shown in the HetSC library [9]. HetSC is an extension library for SystemC to support the modeling of several MoCs. HetSC deals with heterogeneity by the use of configurable converter channels where time and data adaptations are clearly specified. HetSC allows the user to choose if there will be data loss, interpolation or even an exception thrown [10]. These choices are necessary to resolve semantic conflicts among different MoCs.

SystemC-AMS [11] follows a different path, it provides pre-built MoCs allowing co-simulation of continuous and discrete components. SystemC-AMS is the Analog Mixed Signal (AMS) extension for SystemC. As an answer to the heterogeneity problem, SystemC-AMS provides support for MoCs closer to the continuous-time domain such as Linear Signal Flow (LSF) and Electrical Linear Networks (ELN). These are two different ways of representing differential equations on the continuous-time domain, both built on top of a linear differential algebraic equation solver synchronized to the discrete event simulation kernel of SystemC.

SystemC-AMS could be used together with HetSC if one needs to support a wide range of MoCs [12]. In the present work thought we have chosen to target only SystemC-AMS as a first proof-of-concept of our approach which could be easily extended to more MoCs if necessary as discussed in section VI.

C. From SysML/UML to SystemC

Many publications have been devoted to the translation from modeling languages to executable code. Raslan et al. [13] have defined a mapping between SysML and discrete event SystemC in order to raise the abstraction level of electronics designs and speed up the design process. Prevostini et al. [14] proposed a SysML profile to model SoCs and provided an engine to automatically generate SystemC code. By using parametric diagrams they were capable of defining equation constraints, and by using allocations in activity diagrams they managed to co-simulate hardware and software together. Mischkalla et al. [15] came up with a methodology based on an emulated processor using SystemC Transaction Level Modeling (TLM) and reported being capable to automatically synthesize combined hardware and software through a series of code generations and tools synchronizations. They support only the synthesizable subset of SystemC to guarantee that the hardware modeled with SysML is synthesizable.

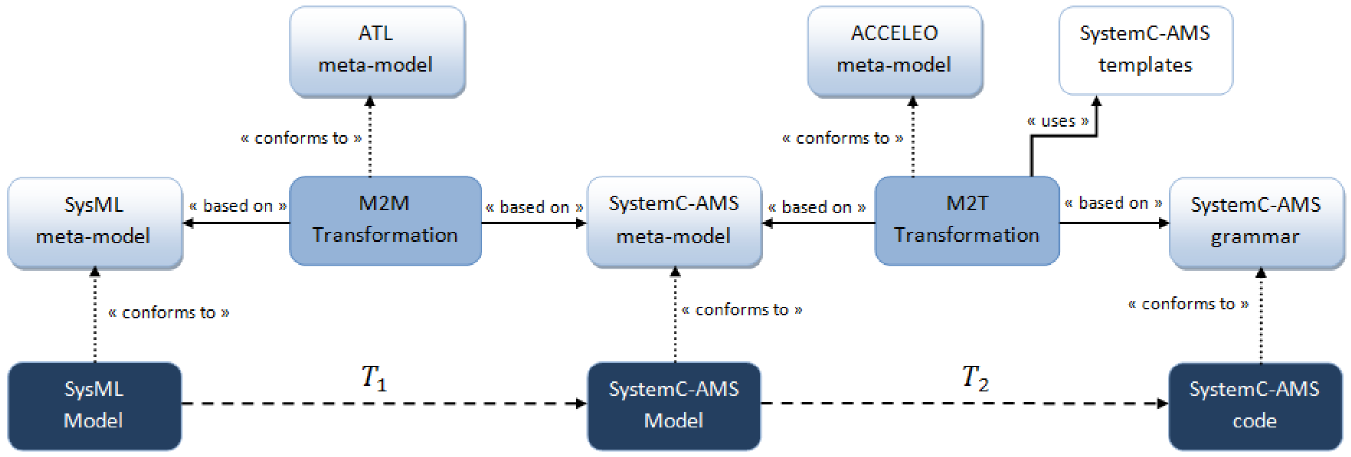

Fig. 1. Transformation chain

All of these approaches have addressed the integration of SysML with the SystemC discrete event simulator or a synthesizable subset of SystemC, but have not considered the intrinsic multi-domain characteristic of heterogeneous systems. We intend to raise the abstraction level of heterogeneous systems designs using SysML diagrams together with a set of semantics definitions for each MoC. That will not only improve systems design comprehensibility with high-level graphical descriptions but also provide executable semantics to SysML diagrams, allowing us to run simulations from the specification models. In the following section we detail our approach by showing an automated way to generate SystemC-AMS code from annotated SysML diagrams.

III. The Approach

Our approach consists in two separated phases. Starting from the SysML model, we first do a model-to-model (M2M) transformation in order to have an equivalent model in the SystemC-AMS language. We then generate SystemC-AMS code through a model-to-text (M2T) transformation using templates of SystemC-AMS.

The M2M transformation takes into consideration the constructions of the input and output languages, thus their meta-models. Since models conform to their meta-models, the transformation can be applied to any instance of the input meta-model. This step is responsible for the translation of every SysML element into its equivalent SystemC-AMS. For example, applying the transformation T1 to a SysML Block composed of several parts results in the creation of a SystemC-AMS module with its corresponding sub-modules.

We have used the Atlas Transformation Language (ATL) [16] to define the M2M transformation. ATL is a language to define model transformations by a set of transformation rules. Being a model itself, the transformation has its own meta-model as well. ATL is based on pattern recognition of *from/to* rules. Every element of the source model that matches any *from* rule triggers the creation of the corresponding *to* element. Therefore, for every SysML element, we have an equivalent SystemC-AMS element.

The example in figure 2 shows a simple rule that will generate a SystemC port (`sc_port`) for every SysML flowport (`sml_port`). This rule will read all attributes from the SysML element, such as name, type and direction, and associate to the equivalent element on the SystemC side.

```
rule Ports {
  from
    sml_port : SYSML!FlowPort
  to
    sc_port : SC!Port (
      name <- sml_port.base_Port.name,
      type <- sml_port.base_Port.type.name,
      direction <- sml_port.direction)
  ...
}
```

Fig. 2. ATL rule for the creation of sc_ports from SysML flowports

In order to define the M2M transformation both input and output meta-models should be available. SysML meta-model has been defined by the Object Management Group (OMG) and it is now an established standard [17] providing a structured definition of the languages elements. Every model written in SysML must conform to its meta-model. SystemC-AMS, on the other hand, has no comparable meta-model reported. Nevertheless, since it is also a language with specific constructions we can define its own meta-model.

SystemC meta-models have been studied in [18] and [19] but were limited to the DE MoC. Based on these previous works, we have added AMS specific constructions and facilities to support multi-formalisms and semantic adaptations. A simplified version with most important elements is shown in figure 3.

Differently from [18], we have separated the definition of an atomic module from a composed module allowing us to diferentiate a user-defined component from a library stardard component e.g. integrators from the LSF formalism

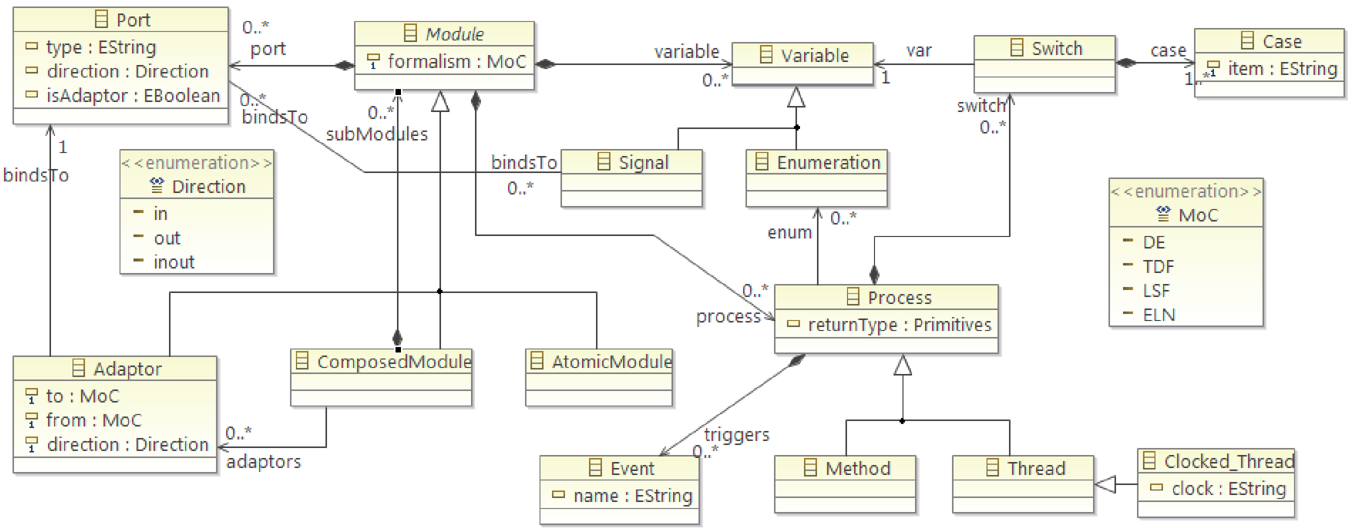

Fig. 3. SystemC-AMS simplified metamodel

or resistances from the ELN formalism. A composed module may contain ports, variables signals and other sub-modules. The later models hierarchical compositions. Some other constructions were inherited from basic C/C++ language, such as the case-switch, necessary to implement a state machine in SystemC.

A module may be modeled using one of the four possible formalisms, either using standard SystemC DE MoC or with any of the specific AMS MoCs, i.e. LSF, ELN or Timed DataFlow (TDF). We take that into account through the use of the *formalism* attribute in the *Module* abstract element. In order to consider the semantic adaptation when crossing the boundaries of different MoCs, we have added an *Adaptor* element that binds to a port.

Adaptors are elements responsable for translating signals from one domain to another. The behavior of an adaptor depends on the combination of input/output MoC and whether it is a producer or consumer of data. We capture those properties in the meta-model by the three attributes *to*, *from* and *direction*. The generated SystemC-AMS code corresponds to the standard adaptor channels available from the SystemC-AMS 1.0 proof-of-concept released by the Fraunhofer Fraunhofer Institute for Integrated Circuits IIS.

Code generation is the second step of our approach. We generate equivalent C++ code from SystemC-AMS model using the ACCELEO engine [20]. ACCELEO is an implementation of the MOF model-to-text language defined by the OMG [21]. ACCELEO is also a transfomation language but the target is text instead of another model. By defining templates for each component of the input meta-model, ACCELEO generates a set of files conforming to the target grammar.

In our approach, ACCELEO scans the input SystemC-AMS model and generates two files for every block, one header with the module definition (equivalent to the black box), and one source file with the implementation of every process.

```
[template public genHeader(m : ComposedModule)]
[file (m.name.concat('.hpp'), false, 'UTF-8')]
...
SC_MODULE([m.name/])
{
  ...
  [for (p : Port | m.port)]
    sc_[p.direction/]<[p.type/]> [p.name/];
  [/for]
...
[/file]
[/template]
```

Fig. 4. ACCELEO: Header generation

In the example of figure 4 we show the creation of each header file when a ComposedModule is found. We define a ComposedModule as a hierarchical element containing other modules. In this example, we want to generate equivalent code of the Module's black box, thus we shall declare every port inside a `SC_MODULE` macro. We do that with a **for** loop that iterates over the sequence of ports of the *Module* ‘m’ and writes equivalent SystemC code. Note that ACCELEO will replace only the code inside the brackets except for internal commands such as **template**, **file**, or **for** loops.

Although code generation is necessary for running simulations, we focus our work on defining concrete semantics to SysML models. We achieve that by using semantic definitions with the help of SysML constraint blocks. The stereotype ”constraintBlock” or simply “constraint” describes constraints on system structures [22]. SysML does not define one language to express constraints. Most will use regular arithmetic expressions to describe relations that can be automatically evaluated. We have chosen to use specific keywords (as we shall demonstrate latter in a case study) to indicate directly in the diagram which MoC is used for each SysML Block.

Our approach for filling the semantics gap in SysML is to define concrete semantics of each MoC along the three dimensions of concurrency, communication and time. We also consider the heterogeneity of multi-paradigm systems and the necessary semantic adaptations at the frontier of different domains. These semantics definitions are implemented by our transformations together with the necessary adaptations. In the following section, we introduce the semantics of two MoCs, i.e. CT and FSM so that simulation of SysML diagrams are free from ambiguous definitions. We also describe briefly the simulation engine on wich the MoCs are based.

IV. A MULTI-PARADIGM SEMANTICS

A. The simulation engine

The execution model is based on the delta-cycle simulation algorithm defined by SystemC's discrete event engine [23]. At the very heart of its engine, the main algorithm is composed of three steps. Evaluate, Update and Time Advancing (also called delta notification). In the evaluation phase, SystemC will run every process but will not propagate data to corresponding signals or ports until every process is executed. The update phase will then synchronize all process by updating signal and ports with previously calculated values on the evaluation phase. This may trigger the engine to re-evaluate some of the process (case of feedback loops) without advancing the simulation time. Finally, when the system's state is stable, time advances until the next scheduled event. This ensures that every node is evaluated before data can propagate and guarantees concurrency of elementary blocks. Concrete semantics is given individually for each MoC.

B. Continuous Time Semantics

Continuous-time models are best expressed using block diagrams. The use of the *internal block diagram* of SysML is suitable to represent hierarchical composition of elements of a system. The CT formalism requires the use of predefined building blocks, such as subtractors, integrators and gain blocks. These primitive blocks are defined in a separated library, shown in figure 5 and are used by the designer to model dedicated transfer functions. The use of the CT formalism is expressed by a SysML constraint block *CT Block* as shown in the example of figure 7.

Concurrency: Every CT block is defined by an equation describing how outputs react to its inputs variations. A CT block shall apply a mathematical function to its input variables every time there is a new sample available on one of its inputs. Mathematical functions can be defined by SysML constraints, as shown for CT building blocks library in figure 5 (for simplicity reasons, only a subset is shown).

Communication is defined by the interpretation of what connectors do. In the case of a CT block, connectors are interpreted as variables of a differential equation. They act as the system memory, saving the state of that system for every snapshot in time.

Time is the independent variable on which some CT blocks rely to apply their mathematical relations. For instance, the gain block has no state and does not depend on time, but the integrator block requires time variations to apply its transfer function.

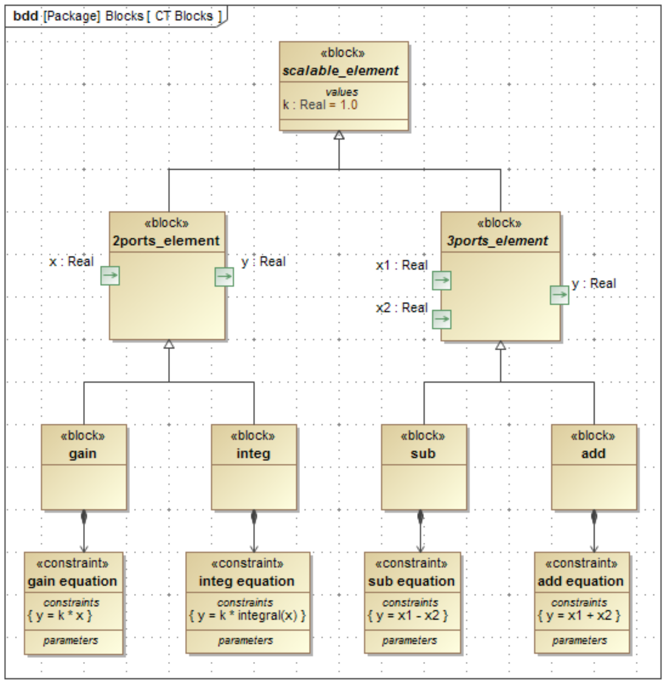

Fig. 5. Continuous Time Building Blocks

C. Finite State Machine Semantics

Finite State Machines have a dedicated diagram in SysML. States are represented by rounded corner rectangles and transitions by arrows. The transition guard is a condition or an event required to change from one state to another. The state invariant represents the control. It takes the form of an equation placed inside the states and produces an output whenever that state is reached.

In our approach we consider FSMs to be untimed, free from implementation details. Indeed, FSMs could have time notations typically synchronous implementations of FSMs on hardware which require a clock input to dictate when guards should be evaluated or outputs produced. We have choosen thought to use a more abstract model of the system thus avoiding implementation-specific constructs.

Concurrency is defined by regions where independent states run concurrently. The most common kind of construction is the or-state, where no concurrency is defined and the system state is defined by the current state itself. A less regular construction is the and-state set. In this case, the system state is defined by a subset of states of independent regions.

Communication is non existent. There is no data flow in a state machine. This kind of diagram is used exclusively to model control.

Time: The notion of time does not exists in a finite state machine. This formalism is driven only by events which do not require a time scale. Semantic adaptation is needed when continuous-time variables are connected to a state machine. In this case, a monitor shall be created for each guard condition to detect threshold crossing and trigger events which are responsible for state changes.

D. Semantic Adaptation

In order to have precise simulations, one has to define not only the semantics of each formalism but also the necessary actions and adaptations if different formalisms are used in the same diagram. This can be achieved by the definition of an adaptor element.

The adaptor is an entity that is bound to a port in order to explicitly adapt data, control and/or time for different formalisms. Our transformation chain chooses apropriate adaptors from the standard SystemC-AMS library depending on the frontier that the port is. For example, if using LSF formalism inside a continuous-time block and the outside environement is of discrete event nature, then a LSF to DE source or sink (`sca_lsf::sca_de::sca_source` or `sca_lsf::sca_de::sca_sink`) should be chosen, depending on the direction of the port.

Some adaptors require the definition of specific attributes. Input ports from DE to CT require the definition of a sampling time-step to guarantee that analog data will be available periodicaly. We ilustrate the use of multi formalims and adaptors in the following case study.

V. Case Study

A. The model

Consider the following example: a vehicle with speed control. This system can be modeled by two blocks: One to model the dynamics of the vehicle and another to model the speed control. We represent the dynamics of the vehicle using an internal block diagram that models the differential equations of the state variables of the system, such as force, acceleration and distance. The control block, on the other hand can be best modeled using state machines. Those are two different formalisms with different semantics. In figure 6 we show the vehicle composed of one part *i_dynamics* typed by the **Dynamics** block and one part *i_control* typed by the **Control** block.

The dynamics block is composed of two integrators and one gain block. They appear as parts of the dynamics block. Note that some blocks have parts that should be initialized with a proper value. In the diagram of figure 6, *init_gain* is one instance of type *gain* with initialized parameters. Other parts will assume default values as defined by their types.

Figure 7 shows the vehicle's dynamics modeled by an internal block diagram with the gain block applying the equation $F = ma$ and two integrators that will compute the speed and distance from the acceleration.

To solve the semantic gap of the internal block diagram, we have added the contraint block *CT Block* with the keyword *useCT*. This implies that semantics defined in section IV should be applied to this diagram. Thus the gain block and both integrators shall apply the mathematical relation defined in figure 5.

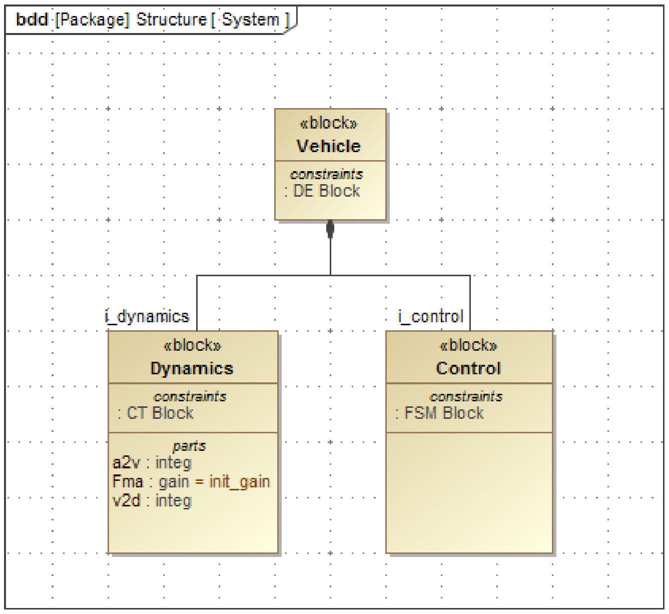

Fig. 6. Vehicle composition

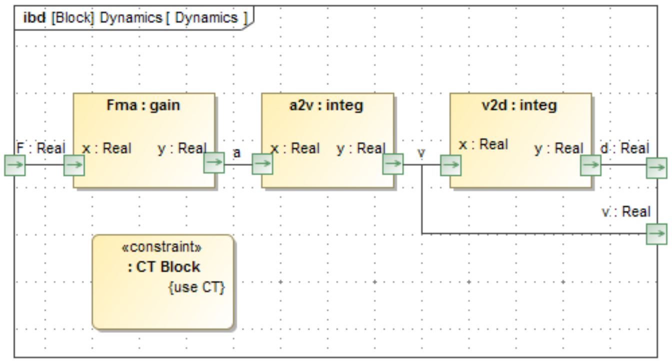

Fig. 7. Vehicle dynamics

The control block is responsible for applying a certain amount of force to the dynamics block, depending on the state of the vehicle. We have modeled it with a State Machine Diagram as we can see in figure 8. The goal is to make the vehicle reach a certain speed, maintain it for a given distance and then stop.

Note that inputs are continuous variables, but inside the control block, we only have events or conditions declared. In this case the semantics of the FSM MoC, as defined in section IV, is used since we have the constraint *FSM Block* with the keyword *useFSM.*

The adaptation is shown in the top-level block, i.e. the internal block diagram of the block Vehicle. In order to define how the state machine interprets analog data and with what precision the inputs are monitored we explicitly annotate in the diagrams that ports are bounded to adaptors using the keyword *isAdaptor* as shown in figure 9.

The declaration of an adaptor is made in the following form: **isAdaptor**(*adaptor type*), where *adaptor type* is a code

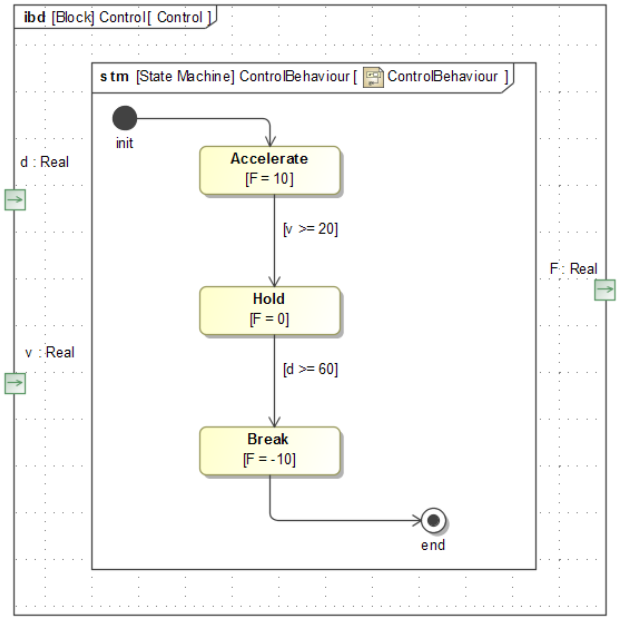

Fig. 8. Vehicle control

for the multi-domain frontier to which the port belongs. In our example, we consider the vehicle to be embedded in the discrete event simulation environment of SystemC. Input port ‘F’ is in the frontier of a DE environment and a CT block. It shall then apply the adaptor type *de2ct*.

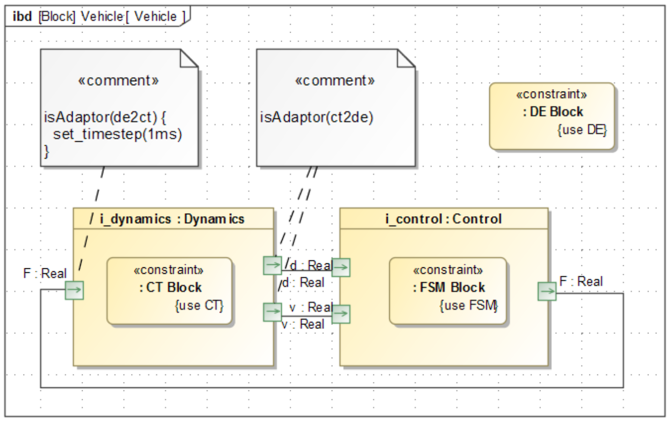

Fig. 9. Vehicle Composed of the dynamics and control

This special case of adaptor will adjust time scale for the CT block because in the DE environment, data won’t be present at all time. In the example, we chose to use periodic sampling by setting the corresponding attribute *timestep* to 1ms. This will create a sample every 1ms at the input port ‘F’ required by the CT block to calculate the outputs ‘v’ and ‘d’, corresponding to speed and distance respectively.

Outputs ‘v’ and ‘d’ apply the inverse adaptor *ct2de*. Contrary to *de2ct* this adaptor will convert data instead of ajusting time. It shall generate an event interpretable by the DE simulator every time a sample is available allowing the FSM to detect with a determined precision (in this case the simulation time step) when events shall trigger its internal guards. The adaptation is a design choice, and the use of adaptors makes it explicit so that different tools can interpret the model in the same way.

B. Results

From our transformations engine, we obtain plain executable SystemC-AMS code. The Continuous Time block was successfully translated to its equivalent LSF model in SystemC-AMS, using base blocks with the same transfer function as defined by the constraints of figure 5. The Finite State Machine was automatically mapped into a two process module with variables `current_state` and `next_state` implementing the classical representation of state machines in SystemC.

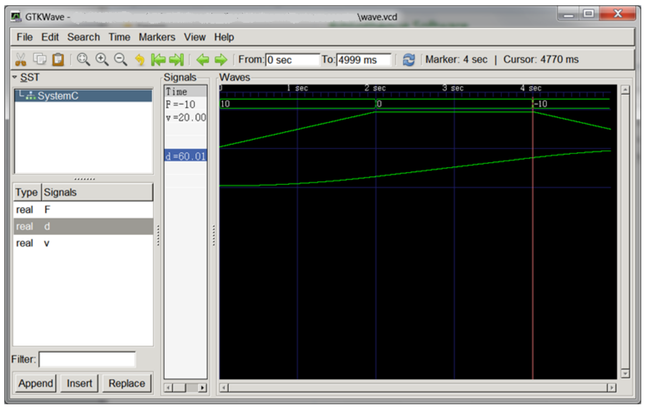

Fig. 10. Results obtained from an automatic code generation

In figure 10, we show the output of the simulation obtained by compiling and running the generated SystemC-AMS code. We can see the force applied to the dynamics model in the first row. It is a signal of discrete nature correctly adapted to work with the continuous time signals ‘v’ and ‘d’. The vehicle accelerates until it reaches a constant speed of 20m/s as specified in the control block of figure 8. After that, control will switch to *hold* state keeping the speed constant until the vehicle reaches 60m slightly after 4 seconds. It finally switches to *break* state until the end of simulation.

The remarkable fact in this simulation is that using only the SysML diagrams we were able to generate the complete executable SystemC-AMS model. Semantics were defined individually for each MoC. Interactions in multi-domain frontiers were strictly described by the adaptors. This approach could be extended to other languages if the meta-model of the target language is available. Identical simulation results would be obtained since the behavior is strictly determined.

VI. Discussions and Conclusions

This two-step technique is a first approach toward a generic intermediary meta-model from wich we could automatically generate code for other languages, e.g. VHDL-AMS or SystemVerilog. With some minor changes, our approach could be extended to support other MoCs. In this case, the user would have to complete the framework with templates of constructions proper to the MoC of interest. If necessary, other languages could be used as well. For example, if the user intents to use Communication Sequential Processes (CSP), HetSC could be a possible candidate. This case results in more changes in the framework such as incrementing the SystemC-AMS meta-model with specific HetSC elements and adding corresponding templates to match HetSC grammar.

The approach has its drawbacks as we try to be as generic as possible. One could claim that since SystemC-AMS provides facilities to model continuous time systems we could benefit of the specific MoCs by defining the use of LSF or ELN directly in the SysML diagrams instead of using a generic MoC CT. This could facilitate the approach by having only one step: the M2T transformation. Again, the choice of the two-step technique allow us to build a generic framework to target the generation of code for other languages.

VII. Conclusion

In this paper we introduce a new approach for simulating multi-domain systems modeled in SysML. We validate the behavior through simulation and we target SystemC-AMS as our execution engine. We address the ambiguity problem of SysML diagrams by assigning concrete semantics (MoCs) to SysML diagrams. In order to solve the semantic adaptation problem, we added the notion of adaptors to SysML based on the existing SystemC-AMS converter channels.

Based on model driven engineering, our transformation framework is capable of generating executable SystemC-AMS code from multi-paradigm SysML diagrams. Our main contribution in this paper was to extend SysML to SystemC code generators by adding (a) concrete semantics to SysML syntactical elements, (b) support for the AMS extension and (c) simulation capabilities to SysML models.

We will continue our work by defining the semantics of other formalisms in SysML diagrams and by improving the specification of testbenches and use cases. We also would like to use semantic verification techniques in our transformation engine in order to verify that SysML models are not only syntactically correct, but also have coherent semantics.

In the future, we wish to apply our strategy to other languages beyond SystemC-AMS. In order to do so, we have considered generalizing our engine with an intermediary abstract semantics to facilitate transformations to other languages. The development of a generic model of computation is an on-going field of research of the GeMoC initiative [24].

References

[1] S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2011.

[2] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 17, no. 12, pp. 1217–1229, 1998.

[3] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains)," *EECS Department, University of California, Berkeley, UCB/EECS-2008-37*, 2008.

[4] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity-the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[5] C. Hardebolle and F. Boulanger, "Modhelx: A component-oriented approach to multi-formalism modeling," in *Models in Software Engineering*. Springer, 2008, pp. 247–258.

[6] IEEE, "Systemc language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–614, 2012.

[7] H.-J. Schlebusch, "Systemc based hardware synthesis becomes reality," in *Euromicro Conference, 2000. Proceedings of the 26th*, vol. 1, 2000, pp. 434 vol.1–.

[8] H. Patel and S. K. Shukla, *SystemC kernel extensions for heterogeneous system modeling: a framework for Multi-MoC modeling & simulation*. Kluwer Academic Pub, 2004.

[9] F. Herrera and E. Villar, "A framework for heterogeneous specification and design of electronic embedded systems in systemc," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 3, p. 22, 2007.

[10] M. Damm, J. Haase, C. Grimm, F. Herrera, and E. Villar, "Bridging mocs in systemc specifications of heterogeneous systems," *EURASIP Journal on Embedded Systems*, vol. 2008, p. 7, 2008.

[11] C. Grimm, M. Barnasconi, A. Vachoux, and K. Einwich, "An introduction to modeling embedded analog/mixed-signal systems using systemc ams extensions," in *DAC2008 International Conference*, 2008.

[12] F. Herrera, E. Villar, C. Grimm, M. Damm, and J. Haase, "Heterogeneous specification with hetsc and systemc-ams: Widening the support of mocs in systemc," in *Embedded Systems Specification and Design Languages*. Springer, 2008, pp. 107–121.

[13] W. Raslan and A. Sameh, "System-level modeling and design using sysml and systemc," in *Integrated Circuits, 2007. ISIC'07. International Symposium on*. IEEE, 2007, pp. 504–507.

[14] M. Prevostini and E. Zamsa, "Sysml profile for soc design and systemc transformation," *ALaRI, Faculty of Informatics University of Lugano via G. Buffi*, vol. 13, no. 5, 2007.

[15] F. Mischkalla, D. He, and W. Mueller, "Closing the gap between uml-based modeling, simulation and synthesis of combined hw/sw systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 2010, pp. 1201–1206.

[16] F. Jouault and I. Kurtev, "Transforming models with atl," *Satellite Events at the MoDELS 2005 Conference*, pp. 128–138, 2006.

[17] OMG, "Systems modeling language (sysml) specification," *OMG standards, formal/2012-06-01*, 2012.

[18] E. Riccobene, A. Rosti, and P. Scandurra, "Improving soc design flow by means of mda and uml profiles," in *3rd Workshop in Software Model Engineering (WiSME 2004)*, 2004.

[19] L. Bondé, C. Dumoulin, and J.-L. Dekeyser, "Metamodels and mda transformations for embedded systems," in *Advances in design and specification languages for SoCs*. Springer, 2005, pp. 89–105.

[20] J. Musset, E. Juliot, S. Lacrampe, W. PIERS, C. BRUN, L. GOUBET, Y. LUSSAUD, and F. ALLILAIRE, "Acceleo user guide," 2006. [Online]. Available: acceleo.org

[21] OMG, "Mof model to text transformation language (mofm2t), 1.0," *OMG standards, formal/08-01-16*, 2008.

[22] T. Weilkiens, *Systems engineering with SysML/UML: modeling, analysis, design*. Morgan Kaufmann, 2011.

[23] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl, "The simulation semantics of systemc," in *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*. IEEE, 2001, pp. 64–70.

[24] GEMOC, "On the globalization of modeling languages," May 2011. [Online]. Available: gemoc.org