A Framework for Unambiguous and Extensible Specification of DSMLs for Cyber-Physical Systems

Gabor Simko[1], David Lindecker[1], Tihamer Levendovszky[1]
Ethan K. Jackson[2], Sandeep Neema[1], Janos Sztipanovits[1]
[1]Vanderbilt University, Nashville, TN
[2]Microsoft Research, Redmond, WA

Abstract—**Increased emphasis on developing model-based design methods for Cyber-Physical Systems (CPS) brings new challenges to the specification of domain specific modeling languages (DSML) and the integration of heterogeneous CPS components. Since CPS are composed of tightly integrated physical and computational components, the modeled domains include both physical and computational systems. Formal specification of physical and computational languages as well as their integration remains an interesting challenge. In this paper we introduce a formal logic based framework for formal specification and simulation, that is supported by the fixed-point logic language FORMULA. As a representative case study, we define both the structural and behavioral semantics for a bond graph language, and demonstrate the reusability and extensibility provided by the approach by extending the language to support hybrid dynamics.**

I. Introduction

Cyber-Physical Systems (CPS) are heterogeneous systems composed of computational and physical components mostly appearing in safety-critical applications, such as automotive, aviation, medical or plant design. Developing safety-critical components is extremely costly, therefore component reusability and extensibility are key design objectives. Component-based engineering is a technique that facilitates designing and building large systems based on these principles.

Whereas the main concern for embedded systems design is the implementation of computational processes that appropriately interact with their known physical and computational environment, in CPS design significant emphasis is placed on the co-design of physical and computational components, and their integration. Furthermore, the vast *diversity* of languages used in CPS brings new challenges to their integration as well. Heterogeneous physical behavior (electrical, mechanical, thermal, and others), CAD models and computational behavior are often modeled using heterogeneous languages, where the *reusability* and *extensibility* of these complicated solutions are crucial for cost effectiveness. Furthermore, understanding the composition of such diverse languages is demanding. In order to facilitate co-design, reusability and to ensure the integrity of the components, we need precise structural and behavioral specifications.

Modeling physical systems calls for high-level modeling languages, which makes Model-Based Engineering (MBE) an often used paradigm for CPS modeling. MBE heavily relies on the usage of Domain-Specific Modeling Languages (DSMLs). DSMLs are languages that are tailored to modeling the important aspects of a specific domain, while using the common terminology known by the experts of the specific domain. Importantly, DSMLs also facilitate the raise of abstraction levels, which in turn increases conciseness, effectiveness and is a fundamental goal in CPS modeling. Often, DSMLs are themselves modeled by a DSML, in which case we refer to such DSMLs as *metamodels*, and the models described by the DSML are the *well-formed models* of the DSML metamodel.

A DSML is a tuple $L = \langle A, C, S, M_S, M_C \rangle$, where C is the concrete syntax, A is the abstract syntax, M_C is a syntactic mapping from A to C, S is the semantic domain, and M_S is the semantic mapping [1]. The concrete syntax is the notation used for representing models, for instance textual syntax or visual syntax. The abstract syntax of a language describes the set of concepts provided by the language, the relations between them, and the well-formedness rules that distinguish the well-formed models from the ill-formed models. The semantic domain together with the semantic mapping defines the formal description behind the concepts and relations: their interpretation in the context of the language.

In general, a model has a structure and a behavior. Accordingly, specification of modeling languages requires support for specifying both structural and behavioral semantics.

Structural semantics describes the meaning of model instances in terms of their structure: all the well-formed models that are defined by the abstract syntax [1]. Structural semantics is described by a mapping from model instances into a two-valued domain, which distinguishes well-formed models from ill-formed models.

Behavioral semantics is represented as a mapping of the model into a mathematical domain that is sufficiently rich for capturing essential aspects of the behavior [2] (such as dynamics). In other words, the explicit representation of behavioral semantics of a DSML requires two distinct components: (i) a mathematical domain and a formal language for specifying behaviors, and (ii) a formal language for specifying transformation between domains.

Different types of behavioral semantics can be distinguished based on the formalism of the description, for instance *denotational semantics* or *operational semantics*. *Denotational semantics* describes the semantics of the language by using some well-defined mathematical formalism. We can specify the denotational semantics of a DSML by defining a semantic domain based on the mathematical formalism and a denotational semantic mapping that transforms models of the DSML to the elements of the semantic domain. *Operational semantics* describes the step-wise execution semantics of a computational language. Formal specification of operational semantics involves defining the transformation that specifies how the system can evolve through its states. For example, Structural Operational Semantics (SOS) is a way to describe operational semantics.

Computational systems are inherently based on discrete events time semantics, and are often defined by operational semantics based on abstract automata notion, or by denotational semantics based on discrete mathematics. Physical modeling languages, however, differ in several ways from computational languages [3]. (i) *Physical structure* is constrained and described by physical laws, thus the structure cannot be arbitrarily shaped according to user demands. The structural semantics of the language must reflect these physical constraints. (ii) The time semantics in physical systems is continuous-time in contrast with discrete events. (iii) *Physical interaction* (behavior) is defined through variable sharing [4] and the *denotational semantics* is represented using differential algebraic equations (DAE).

In the case of physical systems, the behavior is a set of continuous-time trajectories that are usually described by a set of differential algebraic equations (DAE). Thus, a natural choice for the denotational semantic domain of physical systems modeling languages is the notation of DAEs, which indirectly describes the behavior of the represented system. The denotational semantic domain of hybrid systems is more complicated, since it has to deal with continuous-time dynamics and discrete behavior at the same time. Hybrid system semantics is an active research area [5].

Given the many complexities of cyber-physical systems, it is clear that rigorous DSML semantics are essential. In this paper we develop a formal framework for DSMLs arising in CPS that: (i) allows unambiguous specification of denotational semantics for physical modeling languages, (ii) is capable of integrating computational and physical abstractions, (iii) supports extensibility of specifications, (iv) is executable, and facilitates rapid prototyping and simulation of models, (v) allows mathematical reasoning.

In order to support unambiguous specifications and formal reasoning, our framework facilitates *formal* specification based on mathematical logic, while reusability and extensibility of the specifications are naturally supported by the formalism. Furthermore, the specifications are executable, and can be used for fast prototyping and simulations, which is an important feature when developing specifications for large systems. Executability allows systems to be tested before they are built.

We illustrate our framework and its properties described above with a well-known physical modeling language, the bond graph language [6]. Our logic programming environment is a logic programming language called FORMULA [7]. When a design environment adopts a new domain-specific language, the precise meaning of the model elements must be described. Therefore, we firstly develop the formalization for the structure and behavior of a bond graph DSML. Then the design language needs to be modified and extended to meet the requirements of the targeted system design. In order to show how to follow this with extensible specifications, we formalize another variant of the language, a hybrid bond graph language [8]. It also illustrates how the integration between computational and physical parts of the system design can be formally specified. Finally, we give some intuitive guidelines how our framework can be generalized to other CPS domains.

The organization of the paper is as follows: Section 2 describes related work. Section 3 provides an overview of formal semantics, bond graphs, and FORMULA notation. In Section 4 we formalize the structural semantics of a model based bond graph language. The syntax of a differential algebraic equation system in FORMULA, as well as the denotational semantics of the bond graph language are presented in Section 5. In Section 6 we extend our case study to the case of hybrid bond graphs. Finally, we conclude in Section 7.

II. Related work

Previously, significant effort has been devoted to the formalization of various state machine languages. For example [9] used rewriting logic (Maude) to specify the behavioral and structural semantics of state machines. The behavioral semantics of Stateflow is formalized denotationally with lambda-calculus in [10], while the operational semantics of Stateflow is formalized using structural operational semantics (SOS) in [11].

The operational semantics of modeling languages is discussed in [12] by means of graph transformations. A set of graph transformation rules define the formal operational semantics for the language. The operational semantics together with a well-formed model defines a transition system that can be formally verified by model checking.

The behavioral semantics of DSMLs are specified in a translational approach in [1] by means of semantic anchoring and an abstract state machine (ASM) model. A graph transformation language (GReAT) is leveraged to define the semantic mapping from the DSML to an ASM language. Our research goes along the same lines, but instead of using a graph transformation language and a fixed semantic domain, we use a logic based language that brings two advantages: (i) we can specify both the structure and behavior within a single language; (ii) we support larger flexibility.

The translational approach and weaving approach are discussed in [13], which also uses ASM models. Semantic mapping, semantic hooking (anchoring) and semantic meta-hooking are translational approaches, while the weaving approach defines executable specifications directly on the abstract syntax of the languages.

Our current work can be considered the continuation of [14], which describes the application of FORMULA to the specification of DSMLs. Some of the advantages of using FORMULA for semantic anchoring is discussed in [15] and [16].

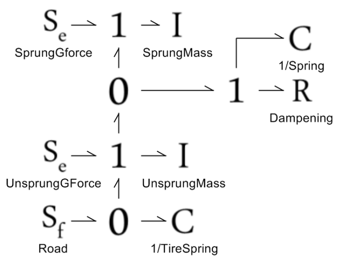

Fig. 1. A quarter car suspension model using the bond graph language.

Our approach is different from previous solutions in the following aspects: (i) we discuss formal specification in the context of CPS systems, (ii) we use an executable fixed-point logic language, which facilitates model conformance checking, model finding, bounded model checking and simulations, (iii) we use the same language for structural and behavioral specifications, which is advantageous in cases when certain properties of the behavior can be elaborated only by understanding the structural semantics of the language.

III. Background

A. Bond graphs

A bond graph is a multi-domain graphical representation of physical systems describing the structure of energy flow in the system [6]. Regardless of the domain – electrical, mechanical, thermal, magnetic or hydraulic – the same representation can be used to describe the flows. Due to their generality, we chose bond graphs to demonstrate formalization of physical dynamics in our framework.

The graph contains nodes and bonds (links) between the nodes (see Fig. 1). Bonds represent the energy exchange between components and is characterized by the power variables: the effort and the flow. The name of these variables is explained by the equation $power = effort \cdot flow$. Furthermore, five node types are distinguished: (i) a dissipative element called resistance **R** having exactly one port, (ii) two storage elements, capacitance **C** and inertia **I**, each having exactly one port, (iii) two power source elements, source of effort **Se** and source of flow **Sf**, each having exactly one port, (iv) two power conserving elements, transformer **TF** and gyrator **GY**, having exactly two ports, (v) two multi-port topological elements, **0-junction** and **1-junction** for composition.

The description indicates that the basic building blocks are different in some ways, in fact, some hints are also provided about the possible meanings (storage, source). However, our formal specification will precisely define these elements. It is well-known that the behavioral semantics of bond graphs can be described using a set of differential algebraic equations (DAE) [6], therefore we will describe the denotational semantics of our bond graph modeling language variant by defining a semantic mapping from our models to sets of DAEs.

B. FORMULA notation

FORMULA is a constraint logic programming tool developed at Microsoft Research [17] based on fixed-point logic and described in detail in [7], [18]. FORMULA expressions are interpreted over a customizable term algebra, that provides ways to formalize sets, relation, functions and other elementary concepts. Furthermore, its fixed-point logic with stratified negation provides an unambiguous and mathematically founded language for describing constrained models over the term algebra. In the paper we use the following notation to describe FORMULA models:

The `domain` keyword specifies a domain (analogous to a metamodel) which is composed of type definitions, data constructors, and rules. A model of the domain consists of a set of *facts* (also called initial knowledge) that are defined using the data constructors of the domain, and the well-formed models of the domain are distinguished from the ill-formed models by the conformance rules.

FORMULA has a complex type system based on built-in types (e.g. `Natural`, `Integer`, `Real`, `String`, `Bool`), enumerations, data constructors, and union types. Enumerations are sets of constants defined by enumerating all their elements, for example `bool ::= {true,false}` denotes the usual 2-valued Boolean type.

Data constructors are constructors for building algebraic terms. Such terms can represent sets, relations, partial and total functions, injections, surjections, and bijections. Data constructor `A ::= new (x:Integer, y:String)` defines term (relation) `A` over pairs of `Integers` and `Strings`, where the optional `x` and `y` are the accessors for the respective values. Data constructor `B ::= fun (x:Integer -> y:String)` defines a partial function (functional relation) `B` from the domain of `Integers` to the codomain of `Strings`. Similarly, `C ::= fun (x:A => y:String)` defines a total function from terms of `A` to `Strings`, `D ::= inj (x:Integer -> y:String)` defines a partial injective function, and `E ::= bij (x:A => y:B)` defines a bijective function between domain `A` and codomain `B`.

While the previous data constructors are used for defining initial facts in models, derived data constructors are used for representing facts derived from the initial knowledge by means of rules. For example, derived data constructor `F ::= (x:Integer, y:String)` defines a term `F` over pairs of `Integers` and `Strings`, which can be used on the left-hand side of rules.

Union types are unions of types in the set-theoretical sense, i.e. the elements of a union type are defined by the union of the elements of the constituent types. FORMULA uses the notation of `T ::= A + B` to define type `T` as the union of type `A` and type `B`.

FORMULA supports the notation of set comprehension in the form of `{head|body}`, which denotes the set of elements formed by `head` that satisfies `body`. Set comprehension is most useful when using built-in operators such as `count`. For instance, given a relation `Pair ::= new (State,State)`, the expression `State(X), n = count({Y|Pair(X,Y)})` counts the number of states paired with state `X`.

Rules allow information to be deduced. They have the form:

$$\texttt{A}_0\texttt{(X) :- A}_1\texttt{(X), } \cdots\texttt{, A}_n\texttt{(X), no B}_1\texttt{(X), } \cdots\texttt{, no B}_m\texttt{(X).}$$

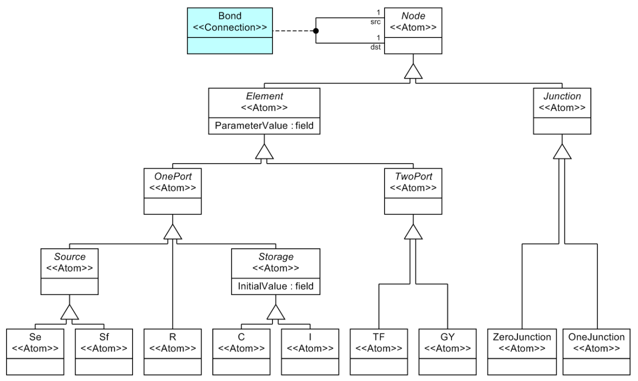

Fig. 2. DSML of bond graphs.

Whenever these is a substitution for `x` where all $\mathtt{A}_1$, $\cdots$, $\mathtt{A}_n$ are derivable and all $\mathtt{B}_1$, $\cdots$, $\mathtt{B}_m$ are not derivable, then $\mathtt{A}_0(X)$ becomes derivable. The use of negation (`no`) is stratified, which implies that rules generate a unique minimal set of derivations, i.e. a least-fix point.

Type constraint `x : A` is true iff variable `x` is of type `A`, while `x is A` is satisfied for all derivations of type `A`. The special symbol _ denotes an anonymous variable that cannot be referred to elsewhere.

The well-formed models of a domain *conforms* to the domain specifications. Each FORMULA domain contains a special `conforms` rule that determines its well-formed models.

Domain composition is supported through the keywords `extends` and `includes`. Both denote the inheritance of all types, data constructors and rules, but while `domain A extends B` ensures that all the well-formed models of `A` are well-formed models of `B`, definition `domain A includes B` might contain well-formed models in `A` which are ill-formed models of `B`.

Finally, FORMULA transformations define rules for creating output models based on input models and parameters. Transformations are also specified using rules where the head terms are the data constructors of the output domain, whereas the body of rules can contain a mixture of the input and output terms and the transformation parameters. The semantics of these transformation rules are simple: if a data constructor term is deducible using the transformation rules, it will be a fact in the output domain.

IV. Bond Graph Structural Semantics

The structure of bond graph represents the energy flows in the physical system. Accordingly, the structure must follow the rules enforced by the physical reality, which is reflected in the structural semantics of the bond graph domain.

We have modeled bond graphs in our metaprogrammable DSML modeling tool Generic Modeling Environment (GME) [19]. First we review the metamodel-based abstract syntax, and then we present the formal structural semantics in FORMULA.

A. GME metamodel of bond graphs

The metamodel of bond graphs in our GME metaprogrammable tool [19] is shown in Fig. 2 with a similar notation to UML class diagrams. In brief, *atom* classes in the metamodel describe classes on the model level, and *connection* classes in the metamodel describe associations on the model level. Abstract classes ($Node$, $Element$, etc.) are denoted with italic names, and basic bond graph elements are inherited from these abstract classes. Abstract classes cannot be instantiated, rather they facilitate the logical grouping of other classes. Triangles denote inheritance, in which case children inherit the attributes of their parent(s). Furthermore, whenever a class can participate in a connection, their derived classes can also be connected using the same connection.

The models of the bond graph domain contains $Node$-s and $Bond$-s between them. The multiplicity of $Bond$ describes that each bond connects exactly one source $Node$ to exactly one destination $Node$. A $Node$ is either an $Element$ or a $Junction$, where an $Element$ is either a $OnePort$ or a $TwoPort$ element, and $Junction$ is either a $ZeroJunction$ or a $OneJunction$. Each $Element$ has a $ParameterValue$ attribute that describes its parameter (e.g. resistance), and is inherited by its children. $OnePort$ elements are either $Source$-s, R-esistors or $Storage$-s. Finally, $Storage$ elements have an $InitialValue$ attribute that describes their initial state.

GME allows well-formedness rules through the usage of OCL constraints (not shown in the figure), for example we have the following constraint attached to $OnePort$ models:

```
context: OnePort
inv: self.attachingConnections(Bond)->size = 1
```

describing the well-formedness rule that $OnePort$ element must have exactly one connecting bond.

B. FORMULA domain for bond graphs

In order to specify the precise semantic mapping between bond graphs and equations, we need to formalize the structure of all the bond graphs. We accomplish this with a term algebraic specification. In general, we can define a one-to-one mapping from the GME metamodel to its FORMULA representation. Each non-abstract class of the metamodel corresponds to a data constructor in the FORMULA domain that can be used to instantiate the class. Further, each class contributes to a union type containing the constructors of the class and all the derived classes. Since abstract classes cannot be instantiated, they only define union types. Finally, OCL and multiplicity constraints are translated to FORMULA rules. Since data constructors and union types cannot have the same name, we distinguish data constructors and corresponding union types by appending $\_c$ to the data constructors.

In particular, the FORMULA class constructors of the bond graph domain are as follows:

```
Se_c ::= new (id:UID).
Sf_c ::= new (id:UID).
R_c  ::= new (id:UID).
C_c  ::= new (id:UID).
I_c  ::= new (id:UID).
```

```
TF_c ::= new (id:UID).
GY_c ::= new (id:UID).
ZeroJunction_c ::= new (id:UID).
OneJunction_c  ::= new (id:UID).
```

Here UID is a unique identifier distinguishing distinct objects of the classes.

Since there are no derived classes from these non-abstract classes, the corresponding union types contain only a single element. For example:

```
Se ::= Se_c.
```

The union types for the abstract classes are specified as follows:

```
Source   ::= Se + Sf.
Storage  ::= C + I.
OnePort  ::= Source + R + Storage.
TwoPort  ::= TF + GY.
Element  ::= OnePort + TwoPort.
Junction ::= ZeroJunction + OneJunction.
Node     ::= Element + Junction.
```

Connections are data constructors with two additional arguments for source and destination. Each bond has exactly one source and one destination, which is captured by the functional relation.

```
Bond_c ::= fun (id:UID -> src:Node,dst:Node).
```

Finally, attributes are defined as primitive data constructors over the previous union type definitions, thus the FORMULA encoding correctly represents the inheritance of attributes. Also note the usage of total functions for ensuring the multiplicity of the attributes.

```
ParameterValue ::= fun (Element => Real).
InitialValue   ::= fun (Storage => Real).
```

Up to this point we have formalized the original GME metamodel with an equivalent FORMULA domain which faithfully represents its structure: classes, abstract classes, attributes and connections.

An important feature of FORMULA is the possibility of defining derived data constructors that can be used as helper functions later. To achieve a concise representation for the bond graph domain, we define the following derived relations:

```
src(A,A.src), dst(A,A.dst) :- A is Bond.
connects(A,X)  :- src(A,X); dst(A,X).
connected(X,Y) :- src(A,X), dst(A,Y).
```

Here src and dst are derived rules describing total functions from bonds to their source and destination nodes, connects is a relation between bonds and its end-points, and connected is a relation describing connected nodes.

Based on these data constructors, the well-formedness rules of bond graphs are shown in Fig. 3. The rules are given in the form of *refutation*, i.e. they describe the ill-formed models. Any model which does not satisfy the ill-formedness rules is considered a well-formed model *conforming* to the domain of bond graphs. In lines 1–10 invalidNode describes the rules of ill-formed nodes, and line 11 describes when a bond graph model conforms to the bond graph domain.

```
 1 // Oneport elements should have exactly one port.
 2 invalidNode :- X is OnePort,
 3                count({Y | connects(Y,X)})!=1.
 4 // Twoport elements should have exactly two ports.
 5 invalidNode :- X is TwoPort,
 6                count({Y | connects(Y,X)})!=2.
 7 // Twoport elements should have welldirected power
 8 // i.e. an incoming and an outgoing bond.
 9 invalidNode :- X is TwoPort, no Bond(_,_,X);
10                X is TwoPort, no Bond(_,X,_).
11 conforms :- no invalidNode.
```

Fig. 3. Well-formedness rules for bond graphs

Lines 1–6 ensure that OnePort elements have exactly one, and TwoPort elements have exactly two connecting bonds. Furthermore, two-port elements must have an incoming and an outgoing bond according to lines 7–10. A bond graph model is structurally valid if it *conforms* to the bond graph domain in line 11, i.e. it does not contain any invalid node or bond.

V. Bond Graph Denotational Semantics

The behavioral semantics of physical systems is best described by differential algebraic equations, and the interconnections of physical components by variable sharing [4]. In our bond graph DSML, we can define the denotational semantics by specifying the translation from the metamodel of bond graphs to the metamodel of differential algebraic equations (DAEs). For this, we need to define first a language for representing DAEs, and then, we need to specify the transformation from the language of bond graphs to the language of DAEs.

A. Differential Algebraic Equation System

In order to be able to map the bond graph elements to a formal domain, not only the bond graph structure must be described in FORMULA, but the formal domain – in our case the equation domain – must also be specified. We define the abstract syntax of a simple language that is capable of describing differential algebraic equations (DAEs) in FORMULA. Such a language defines a set of trajectories: trajectories that satisfy all the equalities. Hence, the behavior described by a set of DAEs is exactly those trajectories that satisfy the conjunction of the equalities.

```
domain DAEs includes UniqueID
{
  term ::= variable + Real + op.
  op   ::= neg + inv + der + mul + sum.
  variable ::= new (id: UID).
  neg    ::= new (term).
  inv    ::= new (term).
  der    ::= new (term).
  mul    ::= new (term, term).
  sum    ::= new (id: UID).
  addend ::= new (sum, term).
  eq     ::= new (lhs:term, rhs:term).
  assign ::= new (lhs:variable, rhs:term).
}
```

A *term* is a (continuous time) *variable*, a *real*, the application of an *op*erator on a term. We define three unary operators *neg*ation, *inv*ersion and *der*ivation, a binary operator *mul*tiplication, and an n-ary operator *sum*mation. *Addend*s of *sums* are represented as relations between *sums* and *terms*. Furthermore, predicate *eq* denotes the equality of the left-hand side and the right-hand side, and assignment operator *assign* denotes a continuous-time unidirectional variable assignment (note that this variable assignment is not the same as in computer languages).

The interpretation of the terms are given as

$$\begin{aligned}
\texttt{variable(i)}^{I} &= v_i(t)\\
\texttt{c}^{I} &= c\\
\texttt{neg(x)}^{I} &= (-\texttt{x}^{I})\\
\texttt{inv(x)}^{I} &= (1/\texttt{x}^{I})\\
\texttt{der(x)}^{I} &= (d\texttt{x}^{I}/dt)\\
\texttt{mul(x,y)}^{I} &= (\texttt{x}^{I}\cdot\texttt{y}^{I})\\
S_x^{I} &= \sum_{y\in Y_x}\texttt{y}^{I}, \text{where } Y_x=\{y \mid \text{Addend}(S_x,y)\}\\
\texttt{eq(x,y)}^{I} &= (\texttt{x}^{I}=\texttt{y}^{I})\\
\texttt{assign(z,y)}^{I} &= (\texttt{z}^{I}:=\texttt{y}^{I})
\end{aligned}$$

Here x, y are terms, z is a variable, $v_i(t)$ is the interpretation of a variable denoting a continuous-time signal function, and c is a real constant. Furthermore, the meaning of operator $:=$ is equality, but extended with the information of causality: the right-hand side term of the assignment is the cause (source) of the left-hand side variable.

B. *Semantic Mapping*

After formally specifying the structure of the bond graph models and a semantic domain, we describe the semantic mapping. Our approach builds on the transformation feature of FORMULA, thus using the same formalism for describing the structural constraints and the transformation. We think this is a unique feature of our approach.

Using the domains of bond graphs and differential algebraic equations, we formalize the denotational semantics of bond graphs. Each bond i defines two variables, flow f_i and effort e_i, indexed by the bond, and each node defines a number of equations on the connected variables. Furthermore, each element has a parameter p_i, which becomes a constant in the DAE system.

In order to make the denotational mapping more concise, we use the following derived data constructors as helpers:

```
b ::= (Bond,variable,variable).
p ::= (Element,Real).
s ::= (Junction,sum).
b(A, variable(ID("e",A.id.id)),
     variable(ID("f",A.id.id))) :- A is Bond.
p(X,Y) :- ParameterValue(X,Y).
s(X,Y) :- X is Junction, Y = sum(X.id).
```

In this snippet `b` is a derived rule describing a total function mapping each bond `A` to a pair of variables – effort and flow –, `p` is an abbreviation for `ParameterValue`, and `s` is a derived rule describing bijective function between `Junction`-s and corresponding `sum`-s.

The rules of denotational semantic mapping is shown in Fig. 4 and in Table I. Note that unless there are conflicts,

```
1  transform DenotationalMapping
2    (in1::BondGraph)
3    returns (out::DAEs)
4  {
5    // Source of effort, equation on effort
6    eq(E,P) :- X : Se, connects(A,X),
       b(A,E,_),p(X,P).
7    // Source of flow, equation on flow
8    eq(F,P) :- X : Sf, connects(A,X),
       b(A,_,F),p(X,P).
9    // Resistance, equation between effort and flow
10   eq(E,mul(P,F)) :- X : R, connects(A,X),
       b(A,E,F),p(X,P).
11   // Capacitance, differential equation between effort and flow
12   eq(der(E),mul(inv(P),F)) :- X : C,
       connects(A,X), b(A,E,F),p(X,P).
13   // Inertia, differential equation between effort and flow
14   eq(der(F),mul(inv(P),E)) :- X : I,
       connects(A,X), b(A,E,F),p(X,P).
15   // Transformer, equations between efforts and flows
16   eq(Ea,mul(P,Eb)),
17   eq(Fb,mul(P,Fa)) :-
18     X : TF, dst(A,X), src(B,X),
       b(A,Ea,Fa),b(B,Eb,Fb),p(X,P).
19   // Gyrator, equations between efforts and flows
20   eq(Ea,mul(P,Fb)),
21   eq(Eb,mul(P,Fa)) :-
22     X : GY, dst(A,X), src(B,X),
       b(A,Ea,Fa),b(B,Eb,Fb),p(X,P).
23
24   // One junction, summation for signed effort, equality for flow
25   eq(S,0)      :- s(X,S), X is OneJunction.
26   addend(S,E)  :- X is OneJunction, dst(A,X),
27                   b(A,E,_), s(X,S).
28   addend(S,neg(E)) :- X is OneJunction,
29                   src(A,X), b(A,E,_), s(X,S).
30   eq(Fa,Fb)    :- X : OneJunction,
31                   connects(X,A), connects(X,B),
32                   b(A,_,Fa), b(B,_,Fb).
33
34   // Zero junction, summation for signed flow, equality for effort
35   eq(S,0)      :- s(X,S), X is ZeroJunction.
36   addend(S,F)  :- X is ZeroJunction, dst(A,X),
37                   b(A,_,F), s(X,S).
38   addend(S,neg(F)) :- X is ZeroJunction,
39                   src(A,X), b(A,_,F), s(X,S).
40   eq(Ea,Eb)    :- X : ZeroJunction,
41                   connects(X,A), connects(X,B),
42                   b(A,Ea,_), b(B,Eb,_).
43 }
```

Fig. 4. Denotational semantics of bond graphs through semantic anchoring

FORMULA automatically uses the concepts of the input model on the right-hand side of the rules, and the concepts of the output model on the left-hand side of the rules. In our case, therefore, we do not have to indicate that `eq` is a concept of `out`, or `connects` is a concept of `in1`.

Line 6 describes the equation for source of efforts. The interpretation for the head of the rule `eq(E,P)` is $e = p$, where the body of the rule describes e and p through the usage of auxiliary variables x and a, where x is a source of effort (Se) and a is the bond whose source is x. Furthermore, `b(A,E,_)` and `p(X,P)` find such variables e and p that e denotes the effort variable of bond a, and p denotes the parameter of node x.

In line 8, the interpretation of `eq(F,P)` is $f = p$. The body of the rule denotes a source of flow x, such that x is the source

TABLE I
CONNECTION BETWEEN BOND GRAPH ELEMENTS, THEIR REPRESENTATIVE EQUATIONS AND LINES OF PRECISE SPECIFICATION IN FIG. 4.

Node	Equation	Lines
Source of effort	$e = p$	6
Source of flow	$f = p$	8
Resistor	$e = p \cdot f$	10
Capacitance	$de/dt = f/P$	12
Inertia	$df/dt = e/P$	14
Transformer	$e_a = p \cdot e_b, f_b = p \cdot e_a$	16–18
Gyrator	$e_a = p \cdot f_b, e_b = p \cdot f_a$	20–22
One-Junction	$\forall a, b.(f_a = f_b), \sum e = 0$	25–28
Zero-Junction	$\forall a, b.(e_a = e_b), \sum f = 0$	31–34

of bond a, where the flow on the bond is f, and the parameter of the source of flow is p.

In line 10, the interpretation of the head of the rule is $e = p \cdot f$. The right hand side of the rule matches resistance x, such that it is the destination of bond a, furthermore e and f respectively denotes the effort and flow on the bond. The parameter of the resistance is p.

In line 12 the head of the rule `eq(der(E),mul(inv(P),F))` denotes $\dot{e} = 1/p \cdot f$ by definition, while the body matches e and f to be the effort and flow variables of a bond whose destination is a capacitance, and p is the capacitance parameter.

Similarly, line 14 defines inertia elements. The interpretation of the head is $\dot{f} = 1/p \cdot e$, where the body of the rule matches inertia x, which is the destination of bond a. Variables e and f are the effort and flow variables across the bond, and p is the parameter of the inertia.

Transformers are defined in lines 16–18 with the equations $e_a = p \cdot e_b$ and $f_b = p \cdot f_a$, where x is a transformer, a and b are its bonds and e_a, e_b, f_a and f_b are the effort and flow variables of the bonds, and p is the modulus of the transformer.

Gyrators are defined in lines 20–22 with the equations $e_a = p \cdot f_b$ and $e_b = p \cdot f_a$, where x is a gyrator, a and b are its bonds and e_a, e_b, f_a and f_b are the effort and flow variables of the bonds, and p is the gyrator modulus.

Zero-junction denotes common efforts on its bonds (line 34), and the flows along its bonds sum to zero (line 31). The addends of the summation are denoted in lines 32–33, which shows the proper signs for the flows. Flows along bonds directed *into* the zero-junction are added with positive signs (line 32), and bonds directed *away* from the zero-junction are added with negative signs (line 33). One-junctions are similarly defined in lines 25–28, only by swapping the role of effort and flow variables.

VI. EXTENSIBILITY

As discussed in the introduction, component reusability and extensibility are especially important in the design of CPS systems. In this section we demonstrate how our framework supports the extension of the specifications.

In order to show how simple it is with our specification approach, we extend our bond graph language with physical domain-specific elements, and with switchable junctions that can be turned on and turned off by an external controller. This case study also illustrates how to provide an interface to cyber parts of the system.

Physical domain specific elements are labeled elements that belong to the physical domain specified by the label. Such elements can be considered typed elements where types range over different physical domains. Switchable junctions extend the bond graph language with discrete modes of operations. Such bond graphs are also known as hybrid bond graphs [8].

A. Physical Domain Specific Elements

While the bond graph language is already an efficient multi-domain physical modeling language, a reasonable requirement is the explicit specification of physical domains for the graph elements. The structure of such a language is further constrained by the compatibility of the different physical domains, for instance electrical and mechanical domain elements are connected through the usage of electric motors and generators.

Using FORMULA, we can quickly augment the bond graph domain with the necessary definitions:

```
Domains ::= {Electrical, Mechanical, Hydraulic}.
NotConverter ::= OnePort + Junction.
DomainType ::= fun (NotConverter => Domains).
ConverterDomainTypes ::= fun (TwoPort =>
                                        Domains,Domains).
```

Here `Domains` is an enumeration of three physical domains, `DomainType` assigns a domain to each bond graph 1-port element and junction, and `ConverterDomainTypes` assigns a source and destination domain to each 2-port element.

Finally, the constraints for valid bond graph connections are based on the domain matching:

```
invalidBond :- A is Bond,
               DomainType(A.src,X),
               DomainType(A.dst,Y), X != Y.
invalidBond :- A is Bond,
               DomainType(A.src,X),
               ConverterDomainTypes(A.dst,Y,_),
               X != Y.
invalidBond :- A is Bond,
               DomainType(A.dst,X),
               ConverterDomainTypes(A.src,_,Y),
               X != Y.
invalidBond :- A is Bond,
               ConverterDomainType(A.src,_,X),
               ConverterDomainTypes(A.dst,Y,_),
               X != Y.
```

Here we rely on type inference to match only the applicable rules. The first rule defines a bond invalid if it connects 1-port or junction elements of different domains, the second and third rules define a bond invalid if it connects a 1-port or junction element to a 2-port element of different type, and the fourth rule defines a bond invalid if it connects two differently typed 2-port elements. If `no invalidBond, no invalidDomainType` is deducible for a well-formed bond graph model, the model is a well-formed model of the extended bond graph metamodel as well.

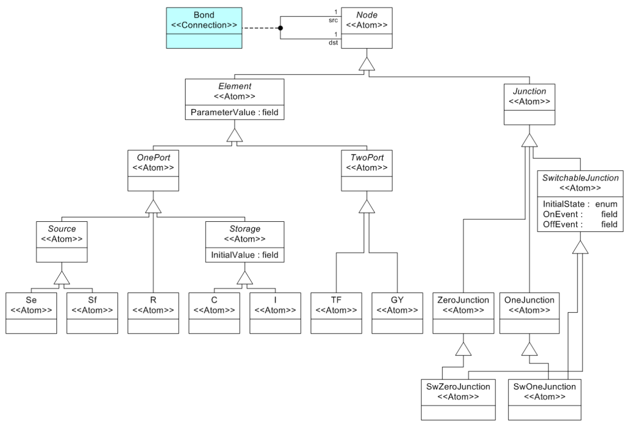

Fig. 5. DSML of hybrid bond graphs.

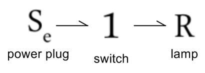

Fig. 6. A hybrid bond graph model.

B. *Structure of Hybrid Bond Graphs*

Fig. 5 shows the hybrid bond graph GME metamodel, that we translate to FORMULA and for which we develop the semantic mapping. An example model is shown in Fig. 6.

In the extended metamodel, switchable junctions are inherited both from the abstract class of `SwitchableJunction` and from the original junction classes, `ZeroJunction` and `OneJunction`. Therefore, switchable junctions inherit the structure and behavior of the original junctions, meanwhile they are extended with attributes for initial state, on events and off events.

The corresponding FORMULA domain extends the original bond graph domain with data constructors for the new junctions:

```
SwZeroJunction_c ::= new (id:UID).
SwOneJunction_c  ::= new (id:UID).
```

Also, union types are updated to reflect the structure of hybrid bond graphs:

```
SwitchableJunction ::= SwZeroJunction
                     + SwOneJunction.
ZeroJunction ::= ZeroJunction + SwZeroJunction.
OneJunction  ::= OneJunction + SwOneJunction.
Junctions ::= ZeroJunction + OneJunction
            + SwitchableJunctions.
```

In the GME metamodel, an enumeration is defined (not shown) with `On` and `Off` values, to represent the initial state of switchable junctions, and the `onEvent` and `offEvent` attributes are defined as string values. The corresponding attributes in FORMULA are as follows:

```
InitialStateEnum ::= { On, Off }.
InitialState ::= fun (SwitchableJunction =>
                      InitialStateEnum).
OnEvent  ::= new (SwitchableJunction, String).
OffEvent ::= new (SwitchableJunction, String).
```

Finally, the multiplicity constraints of the new attributes are defined as rules:

```
// Switchable junctions should have at least one on event.
invalidOnEvent  := X is SwitchableJunction,
                   no OnEvent(X,_).
// Switchable junctions should have at least one off event.
invalidOffEvent := X is SwitchableJunction,
                    no OffEvent(X,_).
```

A hybrid bond graph conforms to the hybrid bond graph metamodel if it conforms to the original bond graph metamodel and does not satisfy the new ill-formedness constraints:

```
BGconforms :- no invalidNode.
conforms :- BGconforms,
            no invalidOnEvent,
            no invalidOffEvent.
```

C. *Behavior of Hybrid Bond Graphs*

We also have to extend our DAE domain with automata behavior to be able to describe the behavior of hybrid bond graphs. The behavior of a switchable junction is defined by an automaton with two modes (`On` and `Off`) and two transitions which are triggered by external events (identified by strings). A switchable junction has a non-empty set of `OnEvent` attributes that define the events which drive the automaton from the `Off` mode to `On` mode. Similarly, `OffEvent` defines the events for the reverse transition. In the case when the hybrid bond graph have several switchable junctions, the behavior is defined as the Cartesian product of the machines defined by the individual switchable junctions.

Similar to the bond graph denotational specification, we wish to define a denotational semantic domain and a semantic mapping which transforms hybrid bond graphs to this semantic domain. In order to support the denotational specification of hybrid bond graphs, we need a semantic domain with support for automata. Therefore, we define the domain of `ParallelHybridAutomata` by extending the DAEs domain by a simple automata model as shown below.

A parallel hybrid automata is composed of parallel running `machine`-s. A `machine` has two `loc`-ations, `on` and `off`. Associated with their `on` and `off` locations, each `machine` has a set of activities, `onAct` and `offAct`, defined as DAE equations. A `switchOn`/`switchOff` transition is a tuple of (m, ev) which describes the transition of machine m from the `off`/`on` location to the `on`/`off` location on event ev. Each `machine` has an `initialLoc`-ation. Finally, since the domain of parallel hybrid automata extends the domain of DAEs, it automatically inherits the data constructors defined in that domain (e.g. variables and equations).

```
domain ParallelHybridAutomata extends DAEs
{
  loc ::= { on, off }.
  label ::= new (String).
  machine ::= new (UID).
  onAct ::= new (m:machine,act:eq).
  offAct ::= new (m:machine,act:eq).
  switchOn ::= new (m:machine,ev:label).
  switchOff ::= new (m:machine,ev:label).
  initialLoc ::= fun (m:machine => init:loc).
}
```

Equivalently, a parallel hybrid automaton is a tuple $H = \langle$ *var, eq, machine, onAct, offAct, label, switchOn, switchOff, initialLoc* $\rangle$.

- A finite set *var* of variables, for which a valuation v is a function that assigns a real value to each variable.
- A finite set *eq* of location-independent activities (DAEs) that constrain the possible valuation of variables.
- A finite set *machine* of machines.
- Location-dependent activities (DAEs) *onAct* and *offAct* that constrain the possible valuation of variables in the on and off state of the machine.
- A finite set *label* of event labels.
- Transition relation *switchOn* where each transition $t_{on} = (m, e)$ defines a transition of machine m from location *off* to location *on* on labeled event e.
- Transition relation *switchOff* where each transition $t_{off} = (m, e)$ defines a transition of machine m from location *on* to location *off* on labeled event e.
- A finite set *initialLoc* of initial locations, one for each machine in the hybrid system.

In the following we describe the semantics of such a hybrid automata built upon the seminal paper [20]. The behavior of a hybrid system is given in terms of the *runs* it produces. At any time instant, the state of the system is completely determined by the current locations $L = \{l^1, l^2, \ldots\}$ of the machines (l^i being the location of machine i), and valuation v. The state can change in two ways: either by a discrete transition between locations, or by continuous evolution of the variables according to the activities.

A run of the system is a sequence $\sigma_0 \rightarrow_{t_0} \sigma_1 \rightarrow_{t_1} \sigma_2 \rightarrow_{t_2} \cdots$ of states where $\sigma_i = (L_i, v_i)$ is a state defined by the current locations of the machines and the variable valuations, and t_i are time instants at which event e_i arrives. The continuous evolution between events are defined by the current location-dependent activities *onAct* and *offAct*, and the location-independent activities *eq*. Given the DAE equations over variables var, the continuous evolution of the system is the set of valuations that satisfy these equations. In the usual representation of DAE equations, valuation v is constrained by the equations $F(\dot{v}(t), v(t), t) = 0$, where t, the time, is the independent variable, and F is defined by the set of activities.

The discrete state transitions at time instants t_i are defined by event e_i and locations $L_{i-1} = \{l^1_{i-1}, l^2_{i-1}, \ldots\}$, where each location l^j_{i-1} denotes the location of machine j at time t_{i-1}. A machine m is fired if there is a transition *switchOn* or *switchOff*, such that $t = (m, e)$ from the current location and is triggered by event e. Function *update*: **state** $\times$ **event** $\rightarrow$ **state** specifies these discrete transitions.

The *update* function of the parallel automaton is given as a parallel update of the individual machines. In the following, up_m: **location** $\times$ **event** $\rightarrow$ **location** is the update function for machine m:

$$update((L_{i-1}, v), e) = ((up_1(l^1_{i-1}, e), \ldots, up_n(l^n_{i-1}, e)), v)$$

$$up_m(l, e) = \begin{cases} off & \text{if } l = on \wedge switchOff(m, e) \\ on & \text{if } l = off \wedge switchOn(m, e) \\ l & \text{otherwise} \end{cases}$$

Finally, we can specify the denotational semantic mapping of hybrid bond graphs by extending the denotational semantic mapping of bond graphs. This also implies that the location-independent activities of the hybrid bond graph are exactly the same equations as the equations of the original bond graph.

Since we extend the original denotational semantic mapping, we reuse the mapping of the original bond graph elements, and we only have to define the structure of the machines and the location-dependent activities. First, we define the mapping of the structures (switchable junctions to machines) as follows:

```
machine(X.id)    :- X is SwJunction.
switchOn(M,E)    :- X is SwJunction, M=machine(X.id),
                    OnEvent(X,E).
switchOff(M,E)   :- X is SwJunction, M=machine(X.id),
                    OffEvent(X,E).
initialLoc(M,on)  :- X is SwJunction,
                     M = machine(X.id),
                     InitialState(X,On).
initialLoc(M,off) :- X is SwJunction,
                     M = machine(X.id)
                     InitialState(X,Off).
```

Second, we define the location-dependent activities for the machines. The behavior of switchable junctions is exactly the same as the behavior of ordinary junctions as long as they are turned on, thus the location-dependent activities `onAct` are empty in the turned-on mode. However, when a switchable junction is turned off, it extends the governing differential algebraic equations as follows:

```
offAct(M,eq(F,0)) :- SwJunction(X), M=machine(X.id),
                     connects(Y,X), b(Y,_,F).
```

With the meaning that there is no flow along the bonds of a turned-off junction.

VII. Conclusion

Precise specifications are essential for the development of CPS systems used in safety-critical scenarios. Semantic mismatches and ambiguities are major issues leading to invalid designs, therefore unambiguous and formal specification of structural and behavioral semantics have far-reaching impact on model-based development of CPS. Usually, DSML specification stops at the level of abstract syntax metamodels (static model), however, in this paper we demonstrated our formal logic based technique for the unambiguous, extensible, reusable and executable specification of both structure and behavior of CPS languages. An important feature is that we used the same logic based language for both the structural and behavioral specifications. Using logic for both the well-formedness rules and the semantic mapping (transformations) facilitates comprehensibility, mathematical reasoning, language reusability and the creation of other language variants as demonstrated in our work.

We illustrated the framework for formal specifications by a bond graph language example, for which we specified the structural and behavioral semantics, as well as presented the extensibility of such specifications by adding physical domain specific elements and hybrid behavior. In order to specify the behavioral semantics of the language, first we defined a semantic domain, the domain of differential algebraic equations, then we specified a denotational semantic mapping from the bond graph domain to our semantic domain. In order to augment the language with hybrid dynamics elements, we also augmented the semantic domain with the notion of parallel automata.

Since our specifications are executable, FORMULA can compute if a model conforms to its metamodel. This can be used for structural verification of models. Further, FORMULA can perform the denotational semantic mapping to obtain the semantic description of a well-formed model. Such a semantic description can be used for rapid prototyping and simulations. For example, by transforming a bond graph model to a set of DAEs, they are readily simulatable by any tools that can solve these equations (e.g. Modelica [21]).

We developed automated tools for generating FORMULA domains from GME metamodels, and FORMULA models from GME models. Further, we can execute the FORMULA transformations to obtain a set of DAEs, and we have an automated tool for generating Modelica simulations based on the DAEs. Using these tools, we have formalized several languages, both from the physical, computational and CPS integration domains. We have specified the structural and behavioral semantics for hybrid bond graphs, ESMoL (an embedded software modeling language [22]) and CyPhyML (Cyber-Physical Modeling Language [23], [24], [25]). These specifications serve as the formal documentation for our languages, they facilitate correct-by-construction modeling, and serve as a reference implementation for model transformations, that can be used for specification-based testing.

Note that in this work the formalization of the semantic domain was not a goal. At the current stage of our research, we use FORMULA for formalizing the structural well-formedness rules, and describing behavioral semantic mappings to different semantic domains. However, it is an important next step to examine in what extent the semantic domains themselves can be formalized using FORMULA.

We have chosen FORMULA as our formalization language, because it is a declarative language with clear semantics, and a good balance between expressiveness and executability. On one side, this allows us to do automated model checking, model finding and conformance testing, on the other side it may turn out that FORMULA is not expressive enough to describe every concept that arises during formalization. Our experiences with FORMULA shows that the language is well-suited for the formalization of structural semantics, operational semantics, and denotational semantic mapping, and in the future we plan to research its applicability in verification of hybrid systems.

Acknowledgement

We are grateful for the useful comments from three anonymous referees. This work was supported by the National Science Foundation under grant number CNS-1035655.

References

[1] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson, "Semantic anchoring with model transformations," in *ECMDA-FA*. Springer, 2005, pp. 115–129.
[2] K. Chen, J. Sztipanovits, and S. Neema, "Compositional specification of behavioral semantics," in *DATE*, 2007, p. 906–911.
[3] E. Lee, "Cyber physical systems: Design challenges," in *ISORC*. IEEE, 2008, pp. 363–369.
[4] J. Willems, "The behavioral approach to open and interconnected systems," *IEEE Control Systems*, vol. 27, no. 6, pp. 46 –99, 2007.
[5] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet, "Non-standard semantics of hybrid systems modelers," *Journal of Computer and System Sciences*, 2011.
[6] D. Karnopp, D. Margolis, and R. Rosenberg, *System dynamics: modeling and simulation of mechatronic systems*. Wiley, New York, 1997.
[7] E. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen, "Components, platforms and possibilities: towards generic automation for MDA," in *EMSOFT*, 2010, p. 39–48.
[8] P. Mosterman and G. Biswas, "A theory of discontinuities in physical system models," *Journal of the Franklin Institute*, vol. 335, no. 3, p. 401–439, 1998.
[9] J. Rivera and A. Vallecillo, "Adding behavior to models," in *EDOC*. IEEE, 2007, pp. 169–169.
[10] G. Hamon, "A denotational semantics for stateflow," in *EMSOFT*. ACM, 2005, pp. 164–172.
[11] G. Hamon and J. Rushby, "An operational semantics for stateflow," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 9, no. 5, pp. 447–456, 2007.
[12] D. Varró, "Automated formal verification of visual modeling languages by model checking," *Software and Systems Modeling*, vol. 3, no. 2, pp. 85–113, 2004.
[13] A. Gargantini, E. Riccobene, and P. Scandurra, "A semantic framework for metamodel-based languages," *Automated software engineering*, vol. 16, no. 3, pp. 415–454, 2009.
[14] E. Jackson, R. Thibodeaux, J. Porter, and J. Sztipanovits, "Semantics of Domain-Specific modeling languages," *Model-Based Design for Embedded Systems*, p. 437, 2009.
[15] D. Balasubramanian and E. Jackson, "Lost in translation: Forgetful semantic anchoring," in *ASE*. IEEE Computer Society, 2009, pp. 645–649.
[16] E. Jackson, W. Schulte, D. Balasubramanian, and G. Karsai, "Reusing model transformations while preserving properties," *Fundamental Approaches to Software Engineering*, pp. 44–58, 2010.
[17] "FORMULA," research.microsoft.com/en-us/projects/formula.
[18] E. Jackson, N. Bjørner, and W. Schulte, "Canonical regular types," *ICLP (Technical Communications)*, p. 73–83, 2011.
[19] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," in *ISP Workshop*, vol. 17, 2001.
[20] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," *Hybrid systems*, pp. 209–229, 1993.
[21] Modelica Association, "Modelica - a unified object oriented language for physical system modeling, language specification, version 3.3," 2012.
[22] J. Porter, G. Hemingway, Nine, H., vanBuskirk, C., Kottenstette, N., Karsai, G., and Sztipanovits, J., "The ESMoL language and tools for High-Confidence distributed control systems design." ISIS, Vanderbilt Univ., Nashville, TN, Tech. Report ISIS-10-109, 2010.
[23] R. Wrenn, A. Nagel, R. Owens, H. Neema, F. Shi, K. Smyth, D. Yao, J. Ceisel, J. Porter, C. vanBuskirk, S. Neema, T. Bapty, D. Mavris, and J. Sztipanovits, "Towards automated exploration and assembly of vehicle design models," in *ASME IDETC/CIE*, 2012.
[24] Z. Lattmann, A. Nagel, J. Scott, K. Smyth, J. Ceisel, C. vanBuskirk, J. Porter, T. Bapty, S. Neema, D. Mavris, and J. Sztipanovits, "Towards automated evaluation of vehicle dynamics in System-Level designs," in *ASME IDETC/CIE*, 2012.
[25] G. Simko, T. Levendovszky, S. Neema, E. Jackson, T. Bapty, J. Porter, and J. Sztipanovits, "Foundation for model integration: Semantic backplane," in *ASME IDETC/CIE*, 2012.