Specification of Cyber-Physical Components with Formal Semantics – Integration and Composition

Gabor Simko, David Lindecker, Tihamer Levendovszky, Sandeep Neema, Janos Sztipanovits

Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN

Abstract. Model-Based Engineering of Cyber-Physical Systems (CPS) needs correct-by-construction design methodologies, hence CPS modeling languages require mathematically rigorous, unambiguous, and sound specifications of their semantics. The main challenge is the formalization of the heterogeneous composition and interactions of CPS systems. Creating modeling languages that support both the acausal and causal modeling approaches, and which has well-defined and sound behavior across the heterogeneous time domains is a challenging task. In this paper, we discuss the difficulties and as an example develop the formal semantics of a CPS-specific modeling language called CyPhyML. We formalize the structural semantics of CyPhyML by means of constraint rules and its behavioral semantics by defining a semantic mapping to a language for differential algebraic equations. The specification language is based on an executable subset of first-order logic, which facilitates model conformance checking, model checking and model synthesis.

Keywords: Cyber-Physical Systems, formalization, formal specification, Model-Based Engineering, heterogeneous composition

1 Introduction

Model-Based Engineering of Cyber-Physical Systems (CPS) needs correct-by-construction design methodologies, hence CPS modeling languages require mathematically rigorous, unambiguous, and sound specifications of their semantics. Cyber-physical systems are software-integrated physical systems often used in safety-critical and mission critical applications, for example in automotive, avionics, chemical plants, or medical applications. In these applications sound, unambiguous, and formally specified modeling languages can help developing reliable and correct solutions.

Traditional systems engineering is based on *causal modeling* (e.g., Simulink), in which components are functional and a well-defined causal dependency exists between the inputs and outputs. It is known that such a causal modeling paradigm is imperfect for physical systems and CPS modeling [31] since physical laws are inherently acausal.

Recently, *acausal modeling* has gained traction and several languages have been introduced for acausal modeling (e.g., Modelica, bond graphs). Every time a new language is introduced, there is a natural demand to extend it to support as many features as possible. Unfortunately, this often leads to enormously large and generic languages, which have many interpretations and variants without a clear, unambiguous semantics. Because of the size of these languages, there is not much hope for complete formalization of their semantics.

A fundamental problem is that generic languages provides support for way more features than a specific problem needs, still they often lack support for some essential functions that would be otherwise needed. Thus, in most cases it is more feasible to use Domain Specific Modeling Languages (DSML), which are designed to support exactly the necessary functions. Additionally, because DSMLs are usually significantly smaller than generic languages, their formal specification is feasible.

In this paper, we focus on the semantic specifications of heterogeneous CPS languages using our CyPhyML DSML as an illustrative example. Our main contribution is an executable specification for CPS languages with a logic-based language for both the structural and behavioral (operational and denotational) semantic specifications, which lends itself to model conformance checking, model finding and Linear Temporal Logic (LTL) model checking. Using the same language for both structural and behavioral semantic specifications is an important step towards better understanding CPS DSMLs and their composition. In previous practices, structure and behavior were formalized in different languages (e.g., OCL and Abstract State Machines) and they were completely separated. Since in our formalism they are represented using the same logic-based formalism, understanding their relations becomes a matter of deductive reasoning. While in this paper we discuss the key concepts for developing the specifications, leveraging these specifications to reason about the connections between structure and behavior remains a matter of future work.

Our working example will be our Cyber-Physical Modeling Language (CyPhyML), an integration language for composing heterogeneous CPS DSMLs. In DARPA's AVM (Adaptive Vehicle Make) program, we required a CPS modeling language that supports the integration of acausal physical modeling, data-flow modeling, CAD models, bidirectional parameter propagation and Design Space Exploration (DSE). While there are several DSMLs that can tackle these problems individually, we needed an integration language to compose them. Therefore, we defined the component-based language CyPhyML, which is capable of representing the integration of heterogeneous components defined in third-party DSMLs. This allows us to compose heterogeneous physical, data-flow and other models designed in external languages and tools such as Modelica, our bond graph language variant or the Embedded Systems Modeling Language (ESMoL).

The organization of the paper is the following: Section 2 describes related work, while Section 3 provides an overview of the background for CPS design, semantics and the formal language that we use. In Section 4 we discuss the metamodel for the compositional sub-language of CyPhyML. Section 5 describes the structural and behavioral semantics of this sub-language and Section 6 discusses the formalization of the integration of third-party DSMLs. Section 7 is devoted to the evaluation and validation of our approach, and Section 8 draws our conclusion.

2 Related Work

The logic-based language FORMULA was first proposed by Jackson [13] as a formal language for specifying the structural semantics of DSMLs and later for specifying their operational semantics [14]. Our research can be considered the continuation of these initiatives. In [29, 30], we used FORMULA for specifying the structural and denotational semantics of a physical modeling language and in [21], we specified the operational semantics of a state-chart language variant. FORMULA provides tools for executing these specifications, in particular they can be used for automated model finding, model conformance checking and LTL model checking.

A different line of research discussed by Rivera [24, 26] uses Maude, an equational logic and term rewriting-based language to specify the operational behavioral and structural semantics of DSMLs. Using Maude's rewriting engine, this representation can be used for LTL model checking, and by leveraging the Real-Time Maude framework it can be used for real-time simulations and analysis [25]. Furthermore, research by Romero [27], Egea [8], and Rusu [28] uses Maude-based formalizations for arguing about model sub-typing, type inference, model conformance and operational semantics of model transformations.

In [6], we introduce a translational approach using the Abstract State Machines (ASM) and a semantic anchoring framework, and in [7], we show how such a semantic anchoring framework can be used for compositional behavioral specifications. Gargantini [10] also introduces an ASM-based semantic framework that includes translational approaches, semantic mapping, semantic hooking and semantic meta-hooking, and a weaving approach for semantic specifications.

Esfahasin [9] uses the Z notation to formally specify the behavioral semantics of an activity-oriented DSML modeled in GME. While Z is not executable, the formal specification provides an unambiguous guideline for automated code generation for their models.

There are several languages for integrating heterogeneous languages, with major emphasis on the composition of heterogeneous computational languages. For example, Ptolemy [11] [12] provides a framework for composing heterogeneous actors described by a variety of Models of Computation (MoC), e.g. finite-state machines, synchronous and dynamic data flows, process networks, discrete events, continuous-time and synchronous-reactive systems. While Ptolemy does support continuous-time dynamics, it lacks support for acausal physical systems modeling.

BIP (Behavior, Interaction and Priority) [2] is a framework that supports the composition of heterogeneous computational systems. The key idea is the separation of component behaviors from component interactions. Such a separation of concerns facilitates the correct composition of components. In [4], the algebra of BIP is formulated, and in [5], the SOS style formalization of glue operators is described.

In this paper, we address the formal semantics of CPS composition languages, which brings additional challenges because of the integration of acausal physical models and causal computational models.

3 Background

3.1 Cyber-Physical Systems

There are significant differences between physical and computational systems. Computational systems are traditionally modeled with the causal modeling approach: components, blocks, software are functional entities, which produce outputs given some inputs. In contrast, physical systems are acausal and the appropriate approach to model them is the acausal modeling approach [31]: interactions are non-directional and there are no input and output ports. Instead, interactions establish simultaneous constraints on the behavior of the connected components by means of *variable sharing*.

For instance, a resistor can be modeled as a two port element, where each port represents a voltage and a current, and the behavior of the resistor is defined by the equations $U_1 - U_2 = R \cdot I_1$ and $I_1 = I_2$. Here, it is unreasonable to talk about the directions of the ports because such a direction is not part of the model: a resistor can be equally driven by a source of current or a source of voltage.

A different problem of CPS modeling is the semantics of time. Physical system models are based on continuous-time (*real* time), while computational systems are inherently discrete-time (e.g., discrete event, periodic discrete time, etc.). The merge of heterogeneous time domains is non-trivial and raises several questions.

If the system uses the notion of events, at any *real* time instant several events may happen simultaneously. To track the causality of these events, we must expand the time domain: super-dense time and non-standard real time [3] have been proposed as expansions of the real time for this purpose. Often, such causally related simultaneous events are the results of the synchronous approach (i.e., the abstraction that computation and communication take zero time).

Another problem is that algebraic loops (loops without delays or integrators) in synchronous systems may have ambiguous semantics: there might be no solutions or several solutions for the system equations. There are several approaches to tackle the problem of algebraic loops: (i) avoid algebraic loops by structural constraints (e.g., Lustre), (ii) do not consider algebraic loops at the design phase, detect problems during simulation (does not support correct-by-construction), (iii) define the least fix-point semantics (Scott semantics) [22].

3.2 Structural and Behavioral Semantics

In general, models represent a structure and associated behaviors. Accordingly, specification of modeling languages requires support for specifying both structural and behavioral semantics [14].

Structural semantics (also known as static semantics) describes the meaning of model instances in terms of their structure [6]. 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 [7]. 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 mapping its syntactic elements to some well-defined (mathematical) semantic domain. The key advantage of denotational semantics is its composability.

Operational semantics describes the step-wise execution of models of the language by an abstract machine. The operational semantics can be formalized as a transformation that specifies how the system evolves through its states.

3.3 FORMULA notation

FORMULA is a constraint logic programming tool developed at Microsoft Research [1] based on first-order logic and fixed-point semantics [15, 16]. It has found many application in Model-Based Engineering such as reasoning about meta-modeling [17] or finding specification errors by constraints [18]. Furthermore, it has been proposed as a formal language for specifying the structural and behavioral semantics of DSMLs as discussed in the related work.

Although we use the newer syntax of FORMULA 2.0, the general principles of the language are unchanged and described in more detail in [15, 16].

The `domain` keyword specifies a domain (analogous to a meta-model) 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 can be used for constructing algebraic data types. Such terms can represent sets, relations, partial and total functions, injections, surjections and bijections. Consider the following type definitions:

```
A ::= new (x:Integer, y:String).
B ::= fun (x:Integer -> y:String).
C ::= fun (x:A => y:String).
D ::= inj (x:Integer -> y:String).
E ::= bij (x:A => y:B).
F ::= (x:Integer, y:String).
```

Data constructor `A` is used for defining `A`-terms by pairing `Integer`s and `String`s, where the optional `x` and `y` are the accessors for the respective values (for example, `A(5,"f")` is an `A`-term). Data constructor `B` is used for defining a partial function (functional relation) from the domain of `Integer`s to the codomain of `String`s. Similarly, `C` is used to define a total function from `A`-terms to `String`s, `D` is used to define a partial injective function, and `E` is used to define a bijective function between `A`-terms and `B`-terms.

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` defines a term over pairs of `Integer`s and `String`s.

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` or `max`. 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:

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

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.

To help writing multiple rules with the same left-hand side term, the semicolon operator is used, whose meaning is logical disjunction. For instance, in `A(X) :- S(X); T(X).` any substitution for `X`, such that `S(X)` or `T(X)` is derivable, makes `A(X)` derivable.

Type constraint `x:A` is true if and only if 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, which 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, which 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 from input models and parameters. Transformations are specified as sets of rules, where the left-hand side terms are the data constructors of the output domain, whereas the right-hand side of the rules can contain a mixture of the terms from the input and output domains, and the transformation parameters. The semantics of these transformation rules is simple: if a data constructor term of the output domain is deducible using the transformation rules, it will be a fact in the output domain.

4 A Cyber-Physical Modeling Language

A CPS modeling language should, at least, contain structures for defining components with physical and computational behaviors, support both acausal and causal modeling and facilitate hierarchical composition. The Cyber-Physical Modeling Language (CyPhyML) we introduce in this section is a minimal language with support for these functions, therefore it serves as a case study for building such languages. The GME meta-model [20] of CyPhyML is shown in Fig. 1.

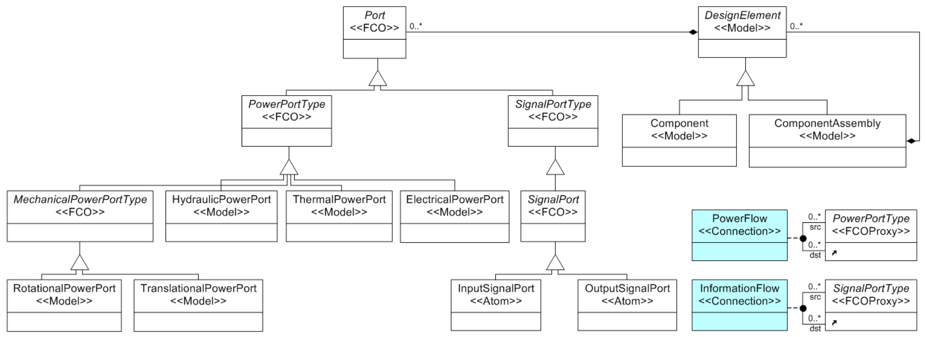
Port <<FCO>>
0..*
DesignElement <<Model>>
0..*
PowerPortType <<FCO>>
SignalPortType <<FCO>>
Component <<Model>>
ComponentAssembly <<Model>>
MechanicalPowerPortType <<FCO>>
HydraulicPowerPort <<Model>>
ThermalPowerPort <<Model>>
ElectricalPowerPort <<Model>>
SignalPort <<FCO>>
PowerFlow <<Connection>>
src
dst
0..*
PowerPortType <<FCOProxy>>
RotationalPowerPort <<Model>>
TranslationalPowerPort <<Model>>
InputSignalPort <<Atom>>
OutputSignalPort <<Atom>>
InformationFlow <<Connection>>
SignalPortType <<FCOProxy>>

Fig. 1. GME meta-model for the composition sub-language of CyPhyML

Components are the main building blocks of CyPhyML. A CPS component represents a physical or computational element with a number of exposed ports. Hierarchical composition is provided by means of component assemblies, which also facilitate component encapsulation and port hiding. There are two types of ports: acausal power ports, denoting the interaction points through which physical energy flows and signal ports, through which causal information flows. CyPhyML is interpreted in continuous (physical) time, thus signals are continuous-time functions. CyPhyML distinguishes several types of power ports, such as electrical power ports, mechanical power ports, hydraulic power ports and thermal power ports.

We can formalize CyPhyML the following way. A CyPhyML model $\mathcal{M}$ is a tuple $\mathcal{M} = \langle C, A, P, \mathit{parent}, \mathit{portOf}, E_P, E_S \rangle$ with the following interpretation:

- C is a set of components,
- A is a set of component assemblies,
- ($D = C \cup A$ is the set of design elements),
- P is the union of the following sets of ports: $P_{rotMech}$ is a set of rotational mechanical power ports, $P_{transMech}$ is a set of translational mechanical power ports, $P_{multibody}$ is a set of multi-body power ports, $P_{hydraulic}$ is a set of hydraulic power ports, $P_{thermal}$ is a set of thermal power ports, $P_{electrical}$ is a set of electrical power ports, P_{in} is a set of continuous time input signal ports, P_{out} is a set of continuous time output signal ports. Furthermore, P_P is the union of all the power ports and P_S is the union of all the signal ports,
- $\mathit{parent} : D \to A^*$ is a containment function, whose range is $A^* = A \cup \{\mathit{root}\}$, the set of design elements extended with a special root element *root*,
- $\mathit{portOf} : P \to D$ is a port containment function, which uniquely determines the container of any port,
- $E_P \subseteq P_P \times P_P$ is the set of *power flow* connections between power ports,
- $E_S \subseteq P_S \times P_S$ is the set of *information flow* connections between signal ports.

We can model these concepts with FORMULA using data constructors and union data types. Thus, the abstract syntax for CyPhyML in FORMULA is the following:

```
C ::= new (id:UID).
A ::= new (id:UID).
D ::= C + A.
P_rotMech ::= new (id:UID).
P_transMech ::= new (id:UID).
...
P_mechanical ::= P_rotMech + P_transMech.
P_power ::= P_mechanical + P_electrical + P_thermal + P_hydraulic.
P_signal ::= P_in + P_out.
P ::= P_power + P_signal.
parent ::= fun (D => A + {root}).
portOf ::= fun (P => D).
Ep ::= new (P_power,P_power).
Es ::= new (P_signal,P_signal).
```

Note that `UID` stands for a unique identifier, which is needed for distinguishing individual members of the sets.

5 Formalization of Semantics

5.1 Structural Semantics

The structural semantics of a language describes the well-formedness rules for its models. We can define the structural semantics of a language using logic rules:

the two-valued semantic domain that distinguishes well-formed and ill-formed models is then equivalent to the deducibility of a special `conforms` constant. To develop the structural semantics of CyPhyML, we define some helper data constructors: Dangling ports are not connected to any other ports:

```
dangling(X) :- X is P_power, no Ep(X,_), no Ep(_,X).
dangling(X) :- X is P_signal, no Es(X,_), no Es(_,X).
```

A distant connection connects two ports belonging to different components, such that the components have different parents, and neither component is parent of the other one:

```
distant(E) :- E is Es(X,Y), portOf(X,PX), portOf(Y,PY), PX != PY,
    Parent(PX,PPX), Parent(PY,PPY), PPX != PPY, PPX != PY, PX != PPY.
distant(E) :- E is Ep(X,Y), portOf(X,PX), portOf(Y,PY), PX != PY,
    Parent(PX,PPX), Parent(PY,PPY), PPX != PPY, PPX != PY, PX != PPY.
```

A power port connection is valid if it connects power ports of same types:

```
validEp(E) :- E is Ep(X,Y), X:P_rotMech, Y:P_rotMech.
...
invalidEp :- E is Ep, no validEp(E).
```

A signal port connection is *invalid* if a signal port receives signals from multiple sources, or an input port is the source of an output port:

```
invalidEs :- E is Es(X,Y), Es(Z,Y), X!=Z.
invalidEs :- E is Es(X,Y), X:P_in, Y:P_out.
```

Note that output ports can be connected to output ports.

Finally, we can express the well-formedness of a CyPhyML model: a model is structurally valid if and only if it does not contain any dangling ports, distant connections and invalid port connections, hence it `conforms` to the domain:

```
conforms :- no dangling(_), no distant(_), no invalidEp, no invalidEs.
```

5.2 Denotational Semantics

The denotational semantics of a language is described by a semantic domain and a mapping that maps the syntactic elements of the language to this semantic domain. In this section, we define a semantic domain for CPS, and specify the semantic mapping from CyPhyML to this domain.

Semantic Domain Continuing our example, the denotational semantics of CyPhyML is described by a semantic mapping from the domain of CyPhyML models to a well-defined mathematical domain, the domain of differential algebraic equations (DAE) extended with periodic discrete-time variables.

Such a semantic domain is reusable: for any CPS language that combines continuous-time physical systems with periodic discrete-time controllers, it can be used as a semantic domain. Furthermore, it facilitates the composition of such languages by establishing a common semantic domain.

We represent the domain of (semi-explicit) differential algebraic equations using the following signature:

```
domain DAEs
{
  term ::= cvar + Real + op.
  op   ::= neg + inv + mul + sum.
  equation ::= eq + diffEq.
  cvar ::= new (UID).
  neg ::= new (term).
  inv ::= new (term).
  mul ::= new (term, term).
  // sum and its addends
  sum    ::= new (UID).
  addend ::= new (sum, term).
  // predicates
  eq     ::= new (term, term).
  diffEq ::= new (cvar, term).
}
```

A `term` is a (continuous time) `var`iable, a `real` number, or the application of an `op`erator on a term. We define two unary operators: `neg`ation and `inv`ersion; a binary operator, `mul`tiplication; and an n-ary operator, `sum`mation. The `addend`s of `sum`s are represented as relations between `sum`s and `term`s. An `equation` is either a predicate `eq` that denotes the equality of the left-hand side and the right-hand side, or a predicate `diffEq` that denotes the differential equation where the derivative of the left-hand side variable equals the right-hand side term.

We extend the DAE domain by adding periodic discrete-time variables, and sample and zero-order hold operators:

```
domain Hybrid extends DAEs
{
  dvar   ::= new (UID,Real,Real).
  sample ::= new (dvar,cvar).
  hold   ::= new (cvar,dvar).
}
```

A hybrid equation extends the differential algebraic equations by periodic discrete-time variables $D \in \mathrm{UID} \times \mathbb{R} \times \mathbb{R}$. A discrete-time variable has a unique identifier, a sampling period p and an initial phase p_0. The discrete-time variable has a well-defined value at *real* times $\{p_0 + n \cdot p \mid n \in \mathbb{N}\}$, everywhere else it is absent.

A model of the hybrid domain is a set of equations E, which represents a set of trajectories over the variables: a trajectory is a function ν that assigns a value to each variable in the system such that $\nu \models E$, i.e., ν simultaneously satisfies all the equations of E. In particular, trajectory ν assigns a real number $\nu(t, x) \in \mathbb{R}$ to each continuous variable x and continuous time t, and ν assigns a value $\nu(t, x) \in \mathbb{R} \cup \perp$ to each discrete variable x, such that $\nu(t, x) = \perp$ when x is absent.

We can extend the valuation function ν to terms: $\nu(t, \texttt{neg(u)}) \overset{def}{=} -\nu(t, \texttt{u})$ and $\nu(t, \texttt{inv(u)}) \overset{def}{=} 1/\nu(t, \texttt{u})$ and $\nu(t, \texttt{mul(u,v)}) \overset{def}{=} \nu(t, \texttt{u}) \cdot \nu(t, \texttt{u})$ and $\nu(t, \texttt{sum(i)}) \overset{def}{=} \sum \nu(t, \texttt{x})$, where the sum is over each `x` for which `addend(sum(i),x)` is a fact.

Finally, the interpretation for the predicates are the following:

$$
\begin{array}{lll}
\nu \models \texttt{eq(u,v)} & \text{if } \nu(t,\texttt{u}) = \nu(t,\texttt{v}) \text{ for all } t \\
\nu \models \texttt{diffEq(u,v)} & \text{if } \frac{d}{dt}(\nu(t,\texttt{u})) = \nu(t,\texttt{v}) \text{ for all } t \\
\nu \models \texttt{sample(u,v)} & \text{if } \begin{cases} \nu(t,\texttt{u}) = \nu(t,\texttt{v}) & \text{if } t = p + n \cdot p_0 \text{ for some } n \in \mathbb{N} \\ \nu(t,\texttt{u}) = \perp & \text{otherwise} \end{cases} \\
\nu \models \texttt{hold(u,v)} & \text{if } \nu(t,\texttt{u}) = \nu(t_0,\texttt{v})
\end{array}
$$

where p, p_0 are the period and initial phase of the discrete variable and t_0 is the greatest upper bound such that $t_0 \leq t$ and $t_0 = p + n \cdot p_0$ for some $n \in \mathbb{N}$.

Semantic Mapping Acausal CPS modeling languages distinguish acausal power ports and causal signal ports. In CyPhyML, each *power port* contributes two variables to the equations, and the denotational semantics of CyPhyML is defined as equations over these variables. *Signal ports* transmit signals with strict causality. Consequently, if we associate a signal variable with each signal port, the variable of a destination port is *enforced* to denote the same value as the variable of the corresponding source port. This relationship is one-way: the value of the variable at the destination port cannot affect the source variable along the connection in question.

Next, we create helper functions to generate unique identifiers for variables and summations in the DAE domain:

```
pV(P,cvar(ID("e",P.id)),cvar(ID("f",P.id))) :- P is P_power.
sV(P,cvar(ID("s",P.id))) :- P is P_signal.
sumName(P,sum(ID("sum",P.id))) :- P is P_power.
```

Relation `pV` maps each power port to a pair of continuous-time variables, `sV` maps signal ports to continuous-time variables and `sumName` assigns a summation operator to each power port. Note the usage of `ID` that is a data constructor for `UID`s; its first argument is a string and its second argument is another `UID`.

Denotational Semantics of Power Port Connections The semantics of power port connections is defined through their transitive closure. Using fixed-point logic, we can easily express the transitive closure of connections as the least fixed point solution for `Ept`:

```
EpT(X,Y) :- Ep(X,Y); Ep(Y,X).
EpT(X,Y) :- EpT(X,Z), Ep(Z,Y), X!=Y;
            EpT(X,Z), Ep(Y,Z), X!=Y.
```

Using `Ept`, we can express the denotational semantics of power ports: power port connections make the effort variables equal and make the flow variables to sum up to zero across the transitively connected power ports (but only those power ports which are contained within a component).

```
eq(S,0), addend(S,F1), addend(S,F2),
eq(E1,E2) :- EpT(P1,P2),
             portOf(P1,C1), C1:Component,
             portOf(P2,C2), C2:Component,
             pV(P1,E1,F1), pV(P2,E2,F2), sumName(P1,S).
```

The explanation, why such a pair of power variables (effort and flow) is used for describing physical connections, is out of scope in this paper, but the interested reader can find a great introduction to the topic in [31].

Denotational Semantics of Signal Port Connections A signal connection path (`EsT`) is a directed path along signal connections. We can use fixed-point logic to find the transitive closure by solving for the least fixed point of `EsT`:

```
EsT(X,Y) :- Es(X,Y).
EsT(X,Y) :- EsT(X,Z), Es(Z,Y).
```

A signal path (SP) is a signal connection path `EsT` such that its end-points are signal ports of components (therefore leaving out any signal ports that are ports of component assemblies).

```
SP(X,Y) :- EsT(X,Y), portOf(X,CX), portOf(Y,CY), CX:C, CY:C.
```

The semantics of signal connection is simply the equality of signal variables:

```
eq(S1,S2) :- EsT(P1,P2), sV(P1,S1), sV(P2,S2).
```

6 Formalization of Language Integration

In the previous section, we have formally defined the semantics of CyPhyML composition, but we have not specified, how components are integrated into CyPhyML. In this section, we develop the semantics for the integration of external languages: a bond graph language and the SignalFlow (ESMoL) language. Note that in the future we can easily augment the list by additional languages (for example, we have developed the integration of a subset of the Modelica language).

Bond Graphs are multi-domain graphical representations for physical systems describing the structure of power flows [19]. Regardless of the domain – electrical, mechanical, thermal, magnetic or hydraulic – the same graphical representation is used to describe the flows. A bond graph contains nodes and bonds (links) between the nodes, where bonds represent the flow of energy between components. This energy flow is represented by power variables: the effort and the flow variables, which are bijectively associated with bonds. Note that these effort and flow variables are different from the effort and flow variables of CyPhyML: they denote different entities in different domains.

Previously, we have introduced a bond graph language along with its formal semantics [30]. In this work, we consider a bond graph language that defines power ports in addition: these are ports through which a bond graph component interacts with its environment. Each power port is connected through exactly one bond, therefore a power port represents a pair of power variables: the power variables of its bond. Our bond graph language also contains output signal ports for measuring effort and flows at bond graph junctions, and modulated bond graph elements that are controlled by input signals through input signal ports.

SignalFlow (ESMoL [23]) is a language and tool-suite for designing and implementing computational and communication models. SignalFlow is based on a periodic time-triggered execution, and its components expose periodic discrete-time signal ports on their interface.

Structural Integration

The role of CyPhyML in the integration process is to establish semantic matching between the languages. Component integration is an error-prone task because of the slight differences between different languages. During the formalization we found the following issues: (i) power ports have different meaning in different modeling languages, (ii) even if the semantics is the same, there are differences in the naming conventions, (iii) the discrete-time signals of SignalFlow must be aligned with the continuous-time CyPhyML signals.

To formalize the integration of external languages, we have to extend CyPhyML with the *semantic interfaces* of these languages. Hence, we need language elements for representing the external models and their containment in CyPhyML, the ports of these external models, and the port mapping between the ports and the CyPhyML ports. The models and their containment are represented by the following data constructors:

```
BondGraphModel  ::= new (id:UID).
SignalFlowModel ::= new (id:UID, rate:Real).
Model ::= BondGraphModel + SignalFlowModel.
ModelContainer ::= fun (Model => Component).
```

Note the second argument of `SignalFlowModel`: since SignalFlow models are periodic, they have a real value describing their period. The interface ports and port mappings are the following:

```
BG_mechanicalRPort ::= new (id:UID).
...
Model_power  ::= BG_powerPort.
Model_signal ::= BG_signalPort + SF_signalPort.
ModelPortOf  ::= fun (Model_power+Model_signal => Model).
ModelPortMap ::= fun (Model_power+Model_signal, String ->
    P_power+P_signal).
```

Here, the second argument of `ModelPortMap` is the *role* of the port mapping. It is used for denoting special port mappings, such as the positive and negative pins of an electrical connector.

Finally, the following elements are added to the well-formedness rules of CyPhyML:

```
// tm(M) denotes that port mapping M is valid (port types are matched)
tm(M) :- M is ModelPortMap(X,_,Y), X:BG_mechanicalRPort, Y:P_rotMech.
...
// invalid, if port mappings are not within same CyPhyMl component:
inv :- ModelPortMap(X,_,Y), ModelPortOf(X,Z), PortOf(Y,W),
       no ModelContainer(Z,W).
// or invalid type matching for any port mapping
inv :- M is ModelPortMap, no tm(M).
// conforms, if both CyPhyML conforms AND port mappings are not ill−formed
conforms :- CyPhyML.conforms, no inv.
```

We also need to extend the definition of our helper functions with the following rules:

```
pV(P,cvar(ID("e",P.id)),cvar(ID("f",P.id))) :- P is Model_power.
sV(P,cvar(ID("s",P.id))) :- P is BG_signalPort.
sV(P,dvar(ID("s",P.id),M.rate,0)) :- P is SF_signalPort,
                                     ModelPortOf(P,M).
```

Note that the SignalFlow ports are converted to discrete-time variables, where the sampling rate is determined by the containing model, and the initial phase defaults to zero.

Bond Graph Integration

For hydraulic and thermal power ports the effort and flow variables of bond graphs and CyPhyML denote the same quantities:

```
eq(E1,E2), eq(F1,F2) :- ModelPortMap(X,_,Y), X:BG_hydraulicThermal,
                        pV(X,E1,F1), pV(Y,E2,F2).
```

In mechanical domains, bond graph efforts denote force and torque and bond graph flows denote velocity and angular velocity. In the CyPhyML language, efforts are position and angular position, flows are force and torque. Therefore, for mechanical power ports, the role of effort and flow is swapped and the derivative of the CyPhyML effort variable is the flow variable of the bond graph:

```
eq(E1,F2), diffEq(E2,F1) :- ModelPortMap(X,_,Y), X:BG_mechanicalPort,
                            pV(X,E1,F1), pV(Y,E2,F2).
```

For the electrical domain, bond graph electrical power ports denote a pair of physical terminals (electrical pins). They are connected to pairs of CyPhyML ports, one to the negative, and the other to the positive pin, which are represented with a plus and minus sign in `ModelPortMap`.

```
eq(F1,F2), eq(F1,F3),
eq(add(E1,E2),E3) :- ModelPortMap(X,"-",Y), ModelPortMap(X,"+",Z),
    X:BG_electricalPort, pV(X,E1,F1), pV(Y,E2,F2), pV(Z,E3,F3).
```

Finally, bond graph and CyPhyML signal ports are semantically matching:

```
eq(U,V) :- ModelPortMap(X,_,Y), sV(X,U), sV(Y,V).
```

SignalFlow Integration

The discrete signals of SignalFlow output ports are converted to continuous-time signals in CyPhyML by means of hold:

```
hold(V,U) :- ModelPortMap(X,_,Y), X:SF_outSignal, sV(X,U), sV(Y,V).
```

Continuous-time signals of CyPhyML input ports are sampled, when mapped to SignalFlow input ports:

```
sample(U,V) :- ModelPortMap(X,_,Y), X:SF_inSignal, sV(X,U), sV(Y,V).
```

7 Semantic Backplane

The presented approach was used for developing the formal specifications for a suite of languages in DARPA's AVM program. These specifications are collectively called the *semantic backplane*. In this section, we provide some details about the size of the languages and the specifications.

The evaluation and validation of the languages are performed through DARPA's on-going FANG challenge (http://vehicleforge.org), during which more than 1000 systems engineers and 200 design teams are using our tools for building vehicle designs. It is interesting to see the complexity of this semantic backplane in terms of its size: CyPhyML, the integration language contains 4121 model elements, which gets compiled into a FORMULA domain with 1635 lines of code (63 enumerated types, 437 union types, 670 primitive data constructors with 2768 attributes). We have developed a code generator that performs this step automatically. The structural and behavioral specifications of the language consists of 1113 lines of code. Furthermore, the complete infrastructure specification adds an additional 2499 lines of code. Altogether, the specifications for the complete system consist of 21 domains, 6 transformations, 647 rules, 262 derived data constructors and 3612 lines of manually written code. On one hand, these numbers indicate the non-trivial size of the project, and on the other hand, it shows that the approach still results in a reasonably compact specification, which – we believe – is comprehensible and relatively easily maintainable.

8 Conclusion

Safety-critical CPS applications call for sound modeling languages, hence we need mathematically rigorous and unambiguous formal specifications for the structural and behavioral semantics of CPS DSMLs. In this paper, we discussed how a logic-based language can be used for specifying both the structural and the denotational behavioral semantics of a CPS language. Our approach has two advantages: (i) we used an executable formal specification language, which lends itself to model conformance checking, model checking and model synthesis; (ii) both the structural and behavioral specifications are written using the same logic-based language, therefore both can be used for deductive reasoning: in particular, structure-based proofs about behaviors become feasible.

So far, we have formally specified the structural and behavioral semantics for CyPhyML, Hybrid Bond Graphs and ESMoL. However, it remains a matter of future work to use these formalizations for model checking, deductive reasoning and correctness proofs.

Acknowledgement

We are grateful for the comments and suggestions received from three anonymous referees, which proved to be very useful. This work was supported by the National Science Foundation under grant number CNS-1035655.

References

1. *FORMULA*. http://research.microsoft.com/en-us/projects/formula.
2. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3 –12, Sept. 2006.
3. A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. Non-standard semantics of hybrid systems modelers. *Journal of Computer and System Sciences*, 78(3):877–910, May 2012.
4. S. Bliudze and J. Sifakis. The algebra of connectors – structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315 –1330, Oct. 2008.
5. S. Bliudze and J. Sifakis. A notion of glue expressiveness for Component-Based systems. In *CONCUR*, page 508–522, 2008.
6. K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic anchoring with model transformations. In *Model Driven Architecture – Foundations and Applications*, volume 3748 of *LNCS*, pages 115–129. Springer, 2005.
7. K. Chen, J. Sztipanovits, and S. Neema. Compositional specification of behavioral semantics. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, page 906–911, San Jose, CA, USA, 2007. EDA Consortium.
8. M. Egea and V. Rusu. Formal executable semantics for conformance in the MDE framework. *Innovations in Systems and Software Engineering*, 6(1-2):73–81, Dec. 2010.
9. N. Esfahani, S. Malek, J. P. Sousa, H. Gomaa, and D. A. Menascé. A modeling language for activity-oriented composition of service-oriented software systems. In *Model Driven Engineering Languages and Systems*, volume 5795, pages 591–605. Springer, 2009.
10. A. Gargantini, E. Riccobene, and P. Scandurra. A semantic framework for metamodel-based languages. *Automated Software Engineering*, 16(3-4):415–454, Apr. 2009.
11. A. Goderis, C. Brooks, I. Altintas, E. Lee, and C. Goble. Composing different models of computation in kepler and ptolemy II. In *Computational Science – ICCS*, volume 4489, pages 182–190. Springer, 2007.
12. A. Goderis, C. Brooks, I. Altintas, E. Lee, and C. Goble. Heterogeneous composition of models of computation. *Future Generation Computer Systems*, 25(5):552–560, May 2009.
13. E. Jackson and J. Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Software and Systems Modeling*, 8(4):451–478, 2009.
14. E. Jackson, R. Thibodeaux, J. Porter, and J. Sztipanovits. Semantics of domain-specific modeling languages. *Model-Based Design for Embedded Systems*, 1:437, 2009.
15. E. K. Jackson, N. Bjørner, and W. Schulte. Canonical regular types. *ICLP (Technical Communications)*, page 73–83, 2011.
16. E. K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. Components, platforms and possibilities: towards generic automation for MDA. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, page 39–48, New York, NY, USA, 2010. ACM.
17. E. K. Jackson, T. Levendovszky, and D. Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In *MoDELS*, pages 653–667, 2011.
18. E. K. Jackson, W. Schulte, and N. Bjørner. Detecting specification errors in declarative languages with constraints. In *MoDELS*, pages 399–414, 2012.

19. D. Karnopp, D. L. Margolis, and R. C. Rosenberg. *System dynamics modeling, simulation, and control of mechatronic systems.* John Wiley & Sons, Hoboken, N.J., 2012.
20. A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, 2001.
21. D. Lindecker, G. Simko, I. Madari, T. Levendovszky, and J. Sztipanovits. Multi-way semantic specification of domain-specific modeling languages. In *ECBS*, 2013.
22. X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, 409(1):110–125, Dec. 2008.
23. J. Porter, G. Hemingway, Nine, H., van Buskirk, C., Kottenstette, N., Karsai, G., and Sztipanovits, J. The ESMoL language and tools for high-confidence distributed control systems design. part 1: Design language, modeling framework, and analysis. Tech. Report ISIS-10-109, ISIS, Vanderbilt Univ., Nashville, TN, 2010.
24. J. E. Rivera, F. Duran, and A. Vallecillo. Formal specification and analysis of domain specific models using maude. *SIMULATION*, 85(11-12):778–792, Aug. 2009.
25. J. E. Rivera, F. Durán, and A. Vallecillo. On the behavioral semantics of real-time domain specific visual languages. In *Rewriting Logic and Its Applications*, volume 6381, pages 174–190. Springer, 2010.
26. J. E. Rivera and A. Vallecillo. Adding behavior to models. In *EDOC*, page 169. IEEE, Oct. 2007.
27. J. R. Romero, J. E. Rivera, F. Duran, and A. Vallecillo. Formal and tool support for model driven engineering with maude. *Journal of Object Technology*, 6(9):187–207, 2007.
28. V. Rusu. Embedding domain-specific modelling languages in maude specifications. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.
29. G. Simko, T. Levendovszky, S. Neema, E. Jackson, T. Bapty, J. Porter, and J. Sztipanovits. Foundation for model integration: Semantic backplane. In *IDETC/CIE*, 2012.
30. G. Simko, D. Lindecker, T. Levendovszky, E. Jackson, S. Neema, and J. Sztipanovits. A framework for unambiguous and extensible specification of DSMLs for cyber-physical systems. In *ECBS*, 2013.
31. J. Willems. The behavioral approach to open and interconnected systems. *IEEE Control Systems*, 27(6):46 –99, 2007.