Interchange Formats for Hybrid Systems: Review and Proposal

Alessandro Pinto[1], Alberto Sangiovanni-Vincentelli[1], Luca P. Carloni[3], and Roberto Passerone[2]

[1] Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720
{apinto, alberto}@eecs.berkeley.edu
[2] Cadence Berkeley Labs, Berkeley, CA 94704
robp@cadence.com
[3] Department of Computer Science, Columbia University in the City of New York, NY 10027-7003
luca@cs.columbia.edu

Abstract. Interchange formats have been the backbone of the EDA industry for several years. They are used as a way of helping the development of design flows that integrate foreign tools using formats with different syntax and, more importantly, different semantics. The need for integrating tools coming from different communities is even more severe for hybrid systems because of the relative immaturity of the field and the intrinsic difficulty of the mathematical underpinnings. In this paper, we provide a discussion about interchange formats for hybrid systems, we survey the approaches used by different tools for analysis (simulation and formal verification) and synthesis of hybrid systems, and we give a recommendation for an interchange format for hybrid systems based on the METROPOLIS metamodel. The proposed interchange format has rigorous semantics and can accommodate the translation to and from the formats of the tools we have surveyed while providing a formal reasoning framework.

1 Introduction

Hybrid systems have proven to be powerful design representations for system-level design in particular for embedded controllers. The term *hybrid* refers to the use of multiple models of computation in a unified framework. Often, hybrid refers to a mix of continuous dynamical systems and finite-state machines even though compositions of heterogeneous systems may be defined in larger semantic domains. The needs for a way of mixing and matching different tools is very much felt because of the relative novelty of this design representation and of the immaturity of the tools available today. There are two camps in the community who deals with hybrid systems: one would prefer to define a common model of computation for hybrid systems that should be used uniformly across different tools, the other pushes for an *interchange format*, i.e., a file, or a set of files, which contains data in a given syntax that is understood by different interacting tools. It is not a database nor a data structure, but a simpler object whose goal is to foster the exchange of data among different tools and research groups. Of course, the approach fostered by the first group has innumerable advantages but it faces an uphill battle with respect to the existing tool vendors or providers such as research groups since embracing this approach would require a substantial rewrite of their tools. The second approach could be strengthened by providing rigorous semantics to the interchange format, thereby allowing a formal analysis of the properties of the translation between different hybrid models.

Our goal in this paper is to provide a survey of models of computation used in a number of tools for the design of hybrid systems and to propose a prototype interchange format based on the Metropolis MetaModel (MMM) that should favor the interaction among the groups involved in hybrid system research and development.

In the U.S., the DARPA MoBIES project had the importance of an interchange format very clear and supported the development of HSIF as a way of fostering interactions among its participants. However, limitations to its semantics make the interchange of data between foreign tools difficult (for example, HSIF does not support some of the features of SIMULINK/STATEFLOW model). To motivate our views, we offer here some considerations about interchange formats that are the result of our experience in the field of Electronic Design Automation (EDA) and of a long history in participating to the formation of standard languages and models for hardware design as well as of Columbus [1], a research project supported by the European Community that spearheaded collaboration across the ocean between European and US research groups.

We believe that an interchange format for tools and designs should:

– support all existing tools, modeling approaches and languages in a coherent global view of the applications and of the theory;
– be open, i.e., be available to the entire community at no cost and with full documentation;
– support a variety of export and import mechanisms;
– support hierarchy and object orientation (compact representation, entry error prevention).

By having these fundamental properties, an interchange format can become the formal backbone for the development of sound design methodologies through the assembly of various tools. The process of moving from the design representation used by tool A to the one used by tool B is structured in two steps: first, a representation in the standard interchange format is derived from the design entry that is used by A, then a preprocessing step is applied to produce the design entry on which B can operate. Notice that tool B may not need all the information on the design that were used by A and, as it operates on the design, it may very well produce new data that will be written into the interchange format but that will not ever be used by A. Naturally, the semantics of the interchange format must be rich enough to capture and "protect" the different properties of the design at the various stages of the design process. This guarantees that there will be no loss going from one design environment to another due to the interchange format itself. The format is indeed a neutral go-between.

In our opinion, HSIF is an excellent model for supporting clean design of hybrid systems but not yet a true interchange format. SIMULINK/STATEFLOW internal format could be a de facto standard but it is not open nor it has features that favor easy import and export. MODELICA has full support of hierarchy and of general semantics that subsumes most if not all existing languages and tools. As such, it is indeed an excellent candidate but it is not open. In addition, all of them have not been developed with the goal of supporting heterogeneous implementations.

On the other hand, the METROPOLIS metamodel (MMM) has generality and can be used to represent a very wide class of models of computation. It has a clear separation between communication and computation as well as architecture and function. While the metamodel itself is perfectly capable to express continuous time systems, there is no tool today that can manage this information in METROPOLIS.

In conclusion, we believe that no approach is mature enough today to recommend its general adoption. However, we believe also that combining and leveraging HSIF, MODELICA, and the METROPOLIS metamodel, we can push for the foundations of a standard interchange format as well as a standard design capture language where semantics is favored over syntax. The discussion of this approach is the main goal of the paper.

2 Preliminaries

This section contains the definitions of hybrid systems and the METROPOLIS metamodel language.

Hybrid Systems. The notion of a hybrid system that has been used in the control community is centered around a particular composition of discrete and continuous dynamics. In particular, the system has a continuous evolution and occasional jumps. The jumps correspond to the change of state in an automaton whose transitions are caused either by controllable or uncontrollable external events or by the continuous evolution. A continuous evolution is associated to each state by means of ordinary differential equations. The structure of the equations and the initial condition may be different for each automaton state. In the sequel, we follow the classic work of Lygeros et al. [2] to define a hybrid system as used in the control literature. In this definition, a hybrid system is a tuple $\mathcal{H} = (\mathbf{Q}, \mathbf{U}_D, X, U, V, \mathbf{S}_C, \mathcal{S}, E, Inv, R, G)$. Without going into the detailed definition of each element of the tuple, we just recall that the triple $(\mathbf{Q}, \mathbf{U}_D, E)$ can be viewed as a Finite State Machine (FSM) having state set $\mathbf{Q}$, inputs $\mathbf{U}_D$ and transitions defined by E. This FSM characterizes the structure of the discrete transitions. A dynamical system is associated to each state and characterized by a set of differential equations. Of particular interest are the mappings Inv, R, G. $Inv : \mathbf{Q} \rightarrow 2^{X \times \mathbf{U}_D \times U \times V}$ is a mapping called *invariant* that is defined over each state of the automaton and states the conditions under which a transition from a state to another in the automaton *must occur.* $R: \ E \times X \times U \times V \to 2^X$ is the reset mapping that defines the initial state of the continuous dynamics after a particular transition has occurred. $G : E \to 2^{X \times U \times V}$ is a mapping called *guard.* G determines the conditions under which a transition *may occur.* The guard and the invariant mappings are complex to analyze with respect to the behavior of the hybrid system. Guards are partly responsible for the non-deterministic behavior of a hybrid system since when a guard allows a transition to occur, the hybrid system may or may not take that transition. The full semantics of a hybrid system are beyond the scope of this paper and can be found in [2].

Metropolis and its Meta-model. The METROPOLIS metamodel [3] is a formalism with precise semantics, yet general enough to support the models of computation [4] proposed so far and, at the same time, to allow the invention of new ones. A behavior can be defined as concurrent occurrences of sequences of *actions.* Some action may follow another action, which may take place concurrently with other actions. The occurrences of these actions constitute the behavior of a system that the actions belong to.

In the metamodel, special types of objects called *process* and *medium* are used to describe computation and communication, respectively. Processes are active objects characterized by a thread that specifies the possible sequence of actions (or better of events, where an event is the beginning or ending of an action) of the process. Medium, instead, are passive objects that offer services and are used for implementing specialized communication protocols. For coordination, one can write formulas in linear temporal logic [5], or use quantity managers to describe a particular algorithmic implementation of constraints. Operationally, a building block called *quantity* is defined in the metamodel language. Its task is to attach tags to events. An execution is then divided in two steps. First, processes issue requests to the quantity managers to annotate their events with particular values of the quantities. Second, the control passes to the quantity managers that order the event depending on the values that have been requested, and decide which requests to grant. In a complex system, multiple quantities could be needed. A quantity manager has to be defined for each quantity. Since their scheduling decisions could depend on each other, the metamodel language provides an interaction mechanism that the user can fully customize to give a specific semantics to the model.

The semantics of the interchange format must be carefully defined to cover all the languages of interest, while still providing efficient and tractable access to subsets corresponding to particular domains of application. The Metropolis metamodel serves this purpose. In fact, states and continuous processes are defined in abstract terms, and can be tailored for the individual needs of a particular model of hybrid behavior. In particular, the mechanisms that determine the operational semantics of the model can be customized by simply encoding the appropriate scheduling policies as the resolution function of the quantity managers dedicated to handling the transition relation and the discretized solution of the continuous dynamics. This flexibility is essential to cleanly, and natively, support different semantic models in a unified environment. In addition, the metamodel is in itself executable, and provides the high level abstract semantics that regulates the scheduling and interaction of processes and quantity managers. Thus, if different models of hybrid behavior are translated into our interchange format, they can also be executed together. Their execution is regulated by the specific choice of managers and resolution functions that are used to glue the system together.

In addition, the full power of the metamodel constraint capabilities and declarative specification can be used to ensure and/or verify that certain properties of interest are satisfied at the border of the domains. This capability is especially important, as it provides a single unified environment for co-simulation and co-analysis.

Note, in particular, that the semantics of interaction between different models of computation is not fixed, but can be defined according to the implementation strategy. The metamodel is first used to define a common semantic domain. Then, the appropriate refinement maps are used to embed each specific model into the common refinement. The semantics of interaction is then the result of applying the metamodel abstract semantics (defined in terms of action automata [3]) to the instances of the models. Thus, different model interactions can be obtained by not only changing the common refinement, but also by playing with the refinement maps. Experimenting with this technique is part of our future work. In particular we aim at integrating different formalisms by showing their individual strengths and weaknesses.

3 A Survey of Languages and Tools for Hybrid Systems

Table 1 shows the approach adopted by each language for modeling the basic hybrid system structure. The first column indicates how discrete and continuous signals are declared in each language. Some languages like CHARON [6] and MODELICA [7] use special type modifiers to specify whether a variable is discrete or continuous. However, the **semantics is different** in the two cases. While CHARON defines a discrete variable constant between two events, hence having derivative equal to zero, the derivative of discrete variables in MODELICA is not defined. Graphical languages like HYVISUAL [8], SIMULINK [9], and SCICOS [10] rely on attributes associated with ports. Type of signals can be automatically inferred during compilation. HYSDEL [11] and CHECKMATE [12] describe the hybrid system as a finite state machine connected to a set of dynamical systems making the interface between discrete and continuous signals fixed and explicit.

Another basic feature is the association of a dynamical system to a specific state of the hybrid automata. HYVISUAL and CHARON seem to have the most intuitive syntax and semantics for this purpose. In HYVISUAL a state of the hybrid automata can be refined into a continuous time system. CHARON allows a mode to be described by a set of algebraic and differential equations. In CHECKMATE, SIMULINK, and HYSDEL a hybrid system is modeled with two blocks: a state machine and a set of dynamical systems. A discrete state tran-

Table 1. Various approaches to modeling hybrid systems

Name	Continuous/Discrete Specification	State/Dynamics Mapping	Continuous/Discrete Interface
CHARON	defined by language modifier	modes refinenement into continuous dynamics	indirect
CHECKMATE	separation between FSMs and dynamical systems	discrete output from FSMs to dynamical systems	event generator first order hold
HYSDEL	real and boolean signals	discrete output from FSMs to dynamical systems	event generator first order hold
HYVISUAL	signal attribute, automatic type detection	state refinement into continuous models	`toContinuous`, `toDiscrete` actors
MODELICA	defined by language modifier	different equation sets depending on events	indirect (`when` statements)
SCICOS	defined by `port` attribute	implemented by connections of event selectors	interaction between continuous/discrete states
SIMULINK	automatic type detection	discrete output from FSMs to dynamical systems	library blocks like zero-order hold.

Table 2. Main features offered by the languages/tools of Table 1

Name	Derivative	Automata Definition	Hierarachy	Object Oriented	Non-Causal Modeling	Algebraic Loops	Dirac Pulses
CHARON	yes	modes of operation	yes	yes	no	no	no
CHECKMATE	yes	STATEFLOW specification	no	no	no	no	no
HYSDEL	discrete differences	logic functions	no	no	no	no	no
HYVISUAL	integration	graphical editor	yes	yes	no	no	no
MODELICA	yes	algorithm sections	yes	yes	yes	no	not yet
SCICOS	integration	network of condit. blocks	yes	no	no	no	no
SIMULINK	derivative and integration	STATEFLOW specification	yes	no	no	no	no

sition can be triggered by an event coming from a particular event-generation block that monitors the values of the variables of the dynamical system. On the other hand, the finite state machine can generate events that are sent to a mode-change that selects a particular dynamics depending on the event. SCICOS implements the automata as an interconnection of blocks whose discrete state can affect the continuous state of blocks implementing the continuous dynamics. Finally, MODELICA provides a set of conditional statements that can change the set of equations describing the continuous state. The last column in Table 1 describes how discrete and continuous signals and blocks interact with each other. CHECKMATE and HYSDEL use an event-generator and a mode-change block. HYVISUAL and SIMULINK provide special library blocks to convert a discrete signal into a continuous and vice versa. In SCICOS, a block can have both continuous and discrete inputs as well as continuous and discrete states.

Discrete states can influence continuous states. Charon and Modelica have special modifiers for distinguishing between discrete and continuous signals. As in all other languages, assignments of one to the other is not possible and can be statically checked (by a simple type checker).

Table 2 shows the features provided by the different tools. All of them support the derivative operator. The specification of the discrete automata has different interpretations. Again, the most intuitive way of describing the discrete automata is implemented by HyVisual and Charon. HyVisual , for instance, has a finite state machine editor where a state machine can be described with bubbles and arcs. Each bubble can then be refined into a continuous time system or into another hybrid system.

Two features are very useful: Object orientation (OO), i.e. the possibility of defining objects and extending them through inheritance and field/method extension, and non-causal modeling, i.e. the possibility of using implicit equations to describe a dynamical system. None of the languages discussed above has a clear definition of the semantics of programs that contain algebraic loops. All of them rely on the simulation engine that, in presence of algebraic loops, either stops with an error message or solves them using specialized algorithms. We believe that a language has to give a meaning to programs containing algebraic loops and the meaning should be independent from the simulation engine.

4 The Interchange Format

In this section, we present a set of requirements for an Interchange Format (IF) and then we proceed to suggest a prototype IF, based on the MMM, by defining its syntax and semantics.

Requirements. An interchange format should be able to capture all the main features of the languages that have been already developed. It has to be a sort of "maximum common denominator" among all hybrid system modeling environments. Specification in the interchange format are not supposed to be written directly by designers. Instead, they should be produced by automatic tools that translate specifications written with other languages into the interchange format. The set of supported features has to be rich enough to guarantee lossless translations. For instance, if the interchange format did not support hierarchy, only flat designs could be described. A translation from one language that supports hierarchy to the interchange format would still be possible but it would inevitably flatten out the design structure, making it impossible to retrieve the original description (the translation process then would be lossy in the sense that the design structure would be lost forever).

We describe the set of features that we believe are essential for an interchange format.

– **Object orientation** is used to group common properties of a set of objects in a base class. It includes the features for defining complex data structures as well as incompletely specified processes. It is possible to extend processes and add/determine part of their behaviors.

- **Hierarchy** is an essential feature for organizing, structuring and encapsulating designs. Flat designs are too complicated to handle because they expose all their complexity in a single view. Even if the interchange format is not supposed to be manipulated directly by designers, it has to retain the original structure.
- **Heterogeneous modeling** is the ability of representing and mixing different models of computation.
- **Refinement** is a language feature for specifying a formal relation between components described at different levels of abstraction. Similarly to Ptolemy, refinement can be used to associate a continuous time dynamics to a discrete state. Since a design can be expressed at different levels of abstraction, formal refinement is definitely an important feature.
- **Implicit equations** are naturally used by designers to describe dynamical systems. An equation represents a constraint on a set of variables.
- **Explicit declaration of discrete states and transitions manager.** A transition manager determines the possible sequence of discrete states of a hybrid automata. Transitions from one state to another, even if defined by the designer, are handled by the simulator, which is hidden. In order for the sequence of states to be preserved across tools, a transition manager should be explicitly described in the interchange format.
- **Explicit declaration of invariant constraints.** Invariants are constraints on the state variables. A set of invariant constraints can be associated with each state of a hybrid system. A specific logic should be supported by the interchange format to specify invariants. METROPOLIS, for instance, defines the logic of constraints (LOC) as a general way of declaring relations among quantities.
- **Explicit non-determinism** must be supported by the interchange format. Languages like the METROPOLIS metamodel have a keyword to specify non-deterministic variables. It is up to the simulation engine to implement non-deterministic choices and return, for instance, one of the possible simulation traces. At the specification level the semantics of a non-deterministic system should include all admissible traces. Non-determinism is important for modeling the environment and for the emergent field of stochastic hybrid systems.
- **Explicit declaration of causality relations and scheduling for variables resolution.** When a system is described with implicit equations, sophisticated techniques are required to understand dependency among variables. After the dependency analysis has been performed, an imperative program can be written that evaluates variables with a specific order. This step is usually hidden but should be explicit in the interchange format since it contributes to the operational semantics of the language.
- **General continuous/discrete interface.** Each modeling environment defines its own communication semantics between continuous and discrete domains. Instead of defining a communication semantics, the interchange for-

mat should provide a set of language primitives that allow the designer to implement any possible communication scheme.

Language syntax. Rather than focusing on object orientation and scoping, we focus on the definition of a few base classes and synchronization statements that should be provided by the interchange format. In order to support heterogeneous modeling, the interchange format should provide a set of basic building blocks that can be used to build several models of computation. The METROPOLIS metamodel provides three basic components: `processes` for doing computation, `media` for communication and `quantity managers` for synchronization and scheduling and starting form this basic classes, we define the following:

– `State` is a process that extends the basic process class. It contains ports representing input and output transitions. These ports are connected to other states and are used to communicate output actions and reset maps.
– `AnalogProcess` is a continuous time process which extends the basic process class. It contains: ports to access external variables that are stored in communication media, and a set of `equation` statements that define the process behavior.
– `TManager` (TM) is the transitions manager which implements the transitions logic of the finite state machine. It defines a `resolve` method which determines the current state.
– `EManager` (EM) is the equation manager. Each equation has a scheduler associated with it. The scheduler defines a `resolve` method that computes unknown values starting from known ones. It uses causality constraints to determine inputs and outputs.
– `ERManager` (ERM) is a manager associated with each dynamical system. It defines a `resolve` method that implements the algorithm to schedule the equation resolution.
– `Transition` is a communication medium used to connect states.
– `AnalogVar` is a communication medium used to connect analog processes.

Besides the basic elements, few other keywords are needed:

– `refine(Object, Netlist)` creates a formal relation between an object and a netlist of components. It is used to build models at different levels of abstraction.
– `invariant{ <formula on state variables> }` is used to specify invariants. `<formula>` is a relation on the state variables.
– `causality(`P`,`var_1` -> `var_2`)` states that var_1 depends on var_2. The two variables must be in the scope of the process P.
– `scheduling(`P_1`,`P_2`,...,`P_N`)` specifies the scheduling order among processes P_1,...,P_N belonging to a dynamical system.

Language semantics. Figure 1 shows a simple example of hybrid system described in the interchange format. Following the METROPOLIS metamodel formalism, we graphically represent processes (analog and states) with squares, communication media with circles and managers with diamonds. The system

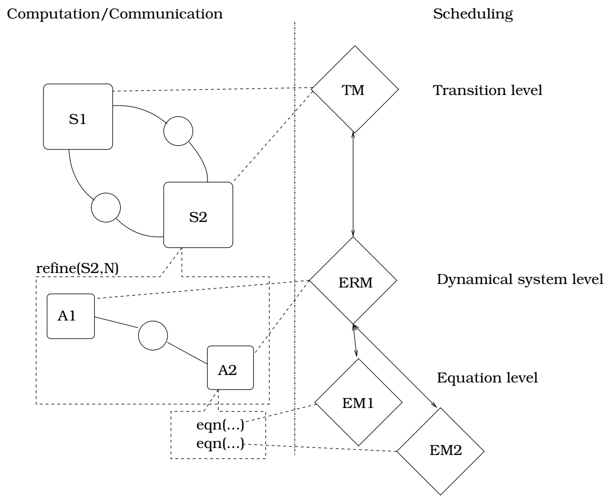

Fig. 1. Graphical representation of an hybrid system using the interchange format

has two discrete states which communicate through media. Each state is refined into a dynamical system (or into another hybrid system). State *S2*, for instance, is refined into a dynamical system composed of two analog processes, *A1* and *A2*. The behavior of an analog process is specified by equations. The netlist is partitioned in the *computation/communication netlist* $\mathcal{CN}$, which represents the structure of the system, and the the *managers netlist* $\mathcal{MN}$ which limits the possible executions of $\mathcal{CN}$ by imposing scheduling constraints. $\mathcal{CN}$ contains processes and communication media. The set of processes $P = \{S, A\}$ is partitioned in the set of states S and the set of analog processes A. Each process behavior is a sequence of events $\{e_i\}$ where an event can have an annotation associated with it (e.g. time).

The execution of a program is defined as a sequence of event vectors $v = [E_S, E_A]$ where $E_S(i)$ is the event executed by the i-th state process and $E_A(i)$ by the i-th analog process. A special event called NOP corresponds to the stalling of a process (refer to [3] for a detailed explanation of the METROPOLIS metamodel semantics). In this setting, an execution is valid if the transition event $E_S(i)$ from state s_i to s_j implies that the set of values associated with events in E_A satisfies the guard conditions defined on the transition. In addition, if the current state is s_i and all events in E_S are equal to NOP, then the set of values associated with events in E_A must satisfy the invariant constraints defined in s_i.

Operationally, the execution consists of a sequence of iterations during which, processes in $\mathcal{CN}$ issue requests to $\mathcal{MN}$ which in turn grants only the requests that are consistent with constraints like guards and invariants. The type of a request depend on the process that issues it. `State` processes issue requests to execute their output transitions while `AnalogProcess` processes issue requests to evaluate their equations sets.

When requests are issued the control is passed to $\mathcal{MN}$ and a coordination between TM, EM and ERM starts to ensure that invariants are satisfied, tran-

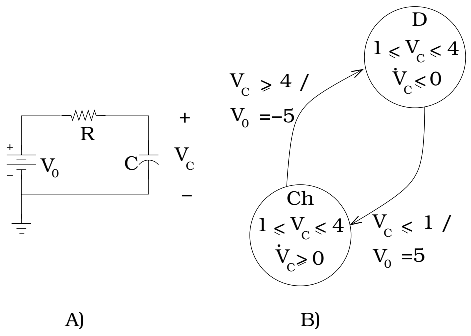

Fig. 2. Simple hybrid system example. A) is the schematic representation of the circuit, B) shows the finite state machine, transitions and invariants

sitions are consistently taken, and equations are evaluated in conformance with causality and scheduling constraints. Note that there could be more than one event vector satisfying all constraints and choosing one is a simulation choice and not a restriction imposed by the language.

Example. Consider the continuous time system of Figure 2. Resistor R and capacitor C are two continuous time processes.

`Capacitor` is a process derived from a general analog process (figure 3). The `AnalogProcess` base class defines special functions for establishing connections to quantity managers. The process has two ports to connect to communication media and read/write variables value. The port type is an interface that declares services that are defined (implemented) by communication media. Note that the ports are not associated with a direction, which implies that the component does not have a causality constraint associated with its description. A resistor is described in the same way but the current/voltage relation is governed by Ohm's law `v = R*i`.

For the description of the system, we refer the reader to Figure 3. The entire continuous time subsystem results from the interconnection of analog processes into a netlist, called `RCCircuit`. Causality constraints and scheduling constraints are specified in this netlist and are used to build the scheduling netlist.

Following is an example corresponding to the charge state of the circuit. Reset maps as well as shared state variables are all accessed through ports. A media has to provide a place to store these variables and also has to implement services to access them. Depending on the implementation of these services, it is possible to customize the communication semantics.

A finite state machine is represented as interconnection of states and transitions. The first part of the netlist instantiates all components including states and communication media. The second part connects states to channels. The last part describes the transitions. Each state declares a set of output transitions that can be connected to the target state in the FSM netlist.

A top level netlist is needed for instantiation of the finite state machine and association of dynamical systems to states. A snippet of the code is also shown

```
process Capacitor extends AnalogProcess {
  parameter double C;
  port AnalogInterface i, v;
  equations {
    i = c * der(v);
  }
}
```

```
process Charge extends State {
  port AnalogChannel v0out, v0in, vc;
  OutTransition vcth(vc >= 4, v0out = -5);
  constraints {
    invariant(vc>=1 && vc <= 4 && der(vc) >= 0);
  }
}
```

```
netlist RCCircuit extends AnalogNetlist {
  port AnalogInterface V0;
  AnalogChannel current, voltagec, voltager = new AnalogChannel(0.0);
  Sub S = new Sub();
  Capacitor C = new Capacitor(1uF);
  Resistor  R = new Resistor(1K);
  connect(S.in1,V0); connect(S.in2,voltagec); connect(S.out,voltager);
  connect(R.v,voltager); connect(R.i,current); connect(C.i,current); connect(C.v,voltagec);
  constraints {
    causality(R,v->i); causality(C,i->v); causality(S,out-> in1 && in2); scheduling(S->R->C);
  }
}

netlist RCFSM extends FSMNetlist {
  Charge ch = new Charge();
  Discharge dch = new Discharge();
  AnalogChannel v0c2d, v0d2c = new AnalogChannel(0.0);
  connect(ch.v0out,v0c2d); connect(ch.v0in,v0d2c);
  connect(dch.v0out,v0d2c); connect(dch.v0in,v0c2d);
  transition(ch.vcth,dch); transition(dch.vcth,ch);
}

netlist Top {
  RCFSM myfsm = new RCFSM();
  refine(myfsm.ch,RCCircuit);
  refine(myfsm.dch,RCCircuit);
  refineconnect(myfsm.ch.v0in, refinementof(myfsm.ch).V0);
  refineconnect(myfsm.dch.v0in, refinementof(myfsm.dch).V0);
  connect(myfsm.ch.vc,refinementof(myfsm.ch).voltagec);
  connect(myfsm.dch.vc,refinementof(myfsm.dch).voltagec);
}
```

Fig. 3. Example of code describing an analog netlist, a state, an FSM and the top netlist

in Figure 3. The top netlist uses the `refine` keyword to associate a dynamical system to a state. A few more connections are specified in the top netlist. First of all we have to connect the reset maps to the dynamical system input. In this case the variable V_0 is an input of the `RCCircuit` netlist. Also we have to connect the variable corresponding to voltage across the capacitor to the state input port. This variable will be checked during simulation for evaluating guards conditions and invariant constraints.

5 Application Scenarios

Consider three hypothetical flows: one where a system is specified and simulated using HyVisual, and then is formally validated using CheckMate. The second is a similar flow where Modelica is used as design entry and simulation tool instead of HyVisual. The third is when a design consists of two parts, one modeled in Modelica and one in HyVisual and we wish to simulate the entire system in Modelica.

To implement these flows, the basic operations are importing into the interchange format from HyVisual and Modelica models and exporting the interchange format into a CheckMate and a Modelica model. Using the interchange format allows a linear number of translations and relative constraints versus a quadratic number of translators if the interchange format is not used.

HyVisual **to Interchange Format.** Translating an HyVisual model into the interchange format is straightforward.

*Computation.*There is a one-to-one correspondence between HyVisual states and state processes in the interchange format. Each state can be refined into another hybrid model or into a dynamical systems. This is possible because the interchange format supports refinement of a generic object into a netlist. Also a dynamical system in HyVisual is constructed as the interconnection of library elements, each of them having a well defined input-output behavior. Each component is mapped into an analog process whose set of equations is defined by the behavior of the respective HyVisual component.

Communication. Each state process has input and output ports representing respectively input and output transitions. Each HyVisual transition from state s_i to s_j is mapped into a transition medium between state process s_i and s_j in the interchange format. If a refinement is associated with the HyVisual transition, then the correspondent transition medium is refined into a netlist.

For each variable v appearing in guard condition c on transition t, there has to be an analog channel from the dynamical system that computes v to the state having t as output transition.

Communication between analog processes in the same dynamical system are implemented by analog communication channels.

Coordination. Each component in HyVisual is causal, i.e. it has inputs and outputs and output values are computed as a function of the inputs. For each analog process a set of causality constraints is added in such a way that outputs depend on the inputs.

Causality and scheduling constraints are used respectively by the equation resolution manager and the equation manager for computing the values of variables at a given time. These two managers in cooperation with the transition manger implement all the algorithms that determine the system operational behavior. For instance, the Runge-Kutta solver can be implemented by the cooperation of ERM and EM. The transition manager, instead, can be implemented so that a request for backtracking is issued to the ERM when a threshold is missed.

Modelica **to Interchange Format.** A Modelica `model` has one or more equation sections that describe its behavior. An equation section can contain `if` and `when` statements whose condition expression generates events. Depending on which event happens, different branches of the conditional statements (and, therefore, a different set of equations) become active. There are several additional restrictions. In particular, the number of variables has always to be equal to the number of equations (non-determinism is avoided by construction).

The translation of a MODELICA program into the interchange format can be done as follows.

Computation. Each MODELICA model is a hybrid system. The number of state processes and the transitions between them are determined by the number of branches resulting from the combination of `if` and `when` statements in the equation sections of the model. Each state process is refined into a dynamical system whose set of equations corresponds to the branch that is active in that state. Since the interchange format supports hierarchy and non-causal modeling (as well as object orientation) translation of the computation aspect does not require special analysis of the original program.

Communication. The only communication mechanism that we should pay attention to is the `connection` primitive that MODELICA defines. Variables involved in a connection are subject to an implicit equation. If the variables are defined as `flow` variables that their sum as to be zero, otherwise they have to be equal. This semantics can be implemented in the interchange format by an analog process that explicitly declares the equations of a connection.

Coordination. Since MODELICA allows non-causal modeling, causality analysis must be performed on the original program to determine the causality and scheduling constraints for each model. In the case of MODELICA functions, this is not needed since inputs and outputs are defined by special keywords.

Interchange Format to MODELICA**.** Each process in the interchange format is mapped to a MODELICA model. If the process is a unrefined analog process then the MODELICA model only contains an equation section where all the equations of the analog process are directly rewritten using the MODELICA syntax.

An interconnection of state processes modeling an hybrid automata (where each process is refined into a dynamical system) is mapped into a MODELICA model presenting a `when` statement with as many branches as states. Each branch is guarded by the guard conditions on the automata transitions. Also each dynamical system that refines a state s is described as a set of equations in the correspondent branch of the when statement representing s.

Note that a model in the interchange format always comes with causality constraints on the variables. Instead of using model then, it is better to use functions in MODELICA that distinguish between input and output.

A composition of hybrid systems in the interchange format is a MODELICA model that instantiates all the systems and interconnects them.

Note that a translation from MODELICA to the interchange format and back will only lose the connection statements since they are translated into analog processes. However, a smart translator could recognize connection processes (e.g., by name) and generate a connection relation among the inputs of the analog connection process.

Interchange Format to CHECKMATE**.** CHECKMATE models hybrid systems using three basic blocks:

- Switched continuous system block (SCSB) of the form $\dot{x} = f(x, \sigma)$ where σ is a discrete variable.
- Polyhedral threshold block (PTHB) whose output is a Boolean variable which is true if $Cx \leq d$ is satisfied. This block represents the conjunction of all guard conditions.
- Finite state machine block (FSMB) that takes the output of PTHB and generates σ.

The function f can be of three types: $x = c$, $\dot{x} = Ax + b$ and $\dot{x} = f(x)$ where f is a non linear function.

Before translating a model from the interchange format to CheckMate, we must verify that the limitations on the guard conditions and on the fields are not violated by the model to be translated. If this is not the case, an error should be notified saying that the target language lacks properties that are required for the description of the original model. After this step, we flatten the design hierarchy. The program in the interchange format has to be analyzed and rewritten in the form of a finite state machine where each state is refined into a dynamical system. The CheckMate FSMB has the same states and transitions of the interchange format one. To build the FSMB we replace each guard condition with a Boolean input coming from the PTHB. The FSMB has an output σ denoting the current state. For each dynamical system d_i which refines state s_i we derive its state space representation. The CheckMate SCSB is the juxtaposition of all this systems and the input σ decides which of this systems is used for computing the state variables. Finally the CheckMate PTHB is obtained as the conjunction of all guard conditions, state variables as inputs and as many Boolean outputs as guard conditions.

6 Conclusions

Hybrid systems are important to a number of applications of great scientific and industrial interest. Being hybrid systems at the same time complex and relatively new, several tools are today available based on different assumptions and modeling strategies. We reviewed the most visible tools for hybrid systems and we presented the case for a novel interchange format based on the Metropolis metamodel (MMM). To do so, we first gave a formal definition of the MMM. We proceeded in listing the requirements and the formal definition of the format. We concluded with examples of use of the interchange format in defining a design flow that includes HyVisual, Modelica and CheckMate to enter the design, simulate it and formally verifying its properties. The interchange format is at this point a proposal, since work still needs to be done to support it with the appropriate debugging and analysis tools and to provide translators to and from the new IF from and to existing tools.

We are confident that a variation of our proposal will be eventually adopted by the community interested in designing embedded systems with particular emphasis on control. We are open to any suggestion and recommendation to improve our proposal.

Acknowledgements. We gratefully acknowledge the discussions on this topic with Janos Stzipanovits of Vanderbilt University and its team, Ed Lee of UC Berkeley, Marika Di Benedetto of University of L'Aquila, Albert Benveniste of INRIA, and the PARADES team (Andrea Balluchi, Alberto Ferrari, Massimo Baleani, Leonardo Mangeruca and Paolo Murrieri). This work has been supported in part by the Columbus Project of the European Community, CHESS ITR, and the GSRC.

References

1. (http://www.columbus.gr/)
2. Lygeros, J., Tomlin, C., Sastry, S.: Controllers for reachability specifications for hybrid systems. In: Automatica, Special Issue on Hybrid Systems. (1999)
3. Balarin, F., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A., Sgroi, M., Watanabe, Y.: Modeling and designing heterogeneous systems. Technical Report 2002/01, Cadence Berkeley Laboratories (2002)
4. Lee, E., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Trans. Comput.-Aided Design Integrated Circuits **17** (1998) 1217–1229
5. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Annual IEEE Symposium on Foundations of Computer Sciences. (1977) 46–57
6. Alur, R., Dang, T., Esposito, J., Hur, Y., Ivancic, F., Kumar, V., Lee, I., Mishra, P., Pappas, G.J., Sokolsky, O.: Hierarchical modeling and analysis of embedded systems. Proceedings of the IEEE (2002)
7. Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. J. Wiley & Sons (2004)
8. Hylands, C., Lee, E.A., Liu, J., Liu, X., Neuendorffer, S., Zheng, H.: Hyvisual: A hybrid system visual modeler. Technical Report UCB/ERL M03/1, UC Berkeley (2003) available at `http://ptolemy.eecs.berkeley.edu/hyvisual/`.
9. Dabney, J.B., Harman, T.L.: Mastering Simulink. Prentice Hall (2003)
10. Nikoukhah, R., Steer, S.: SCICOS A dynamic system builder and simulator user's guide - version 1.0. Technical Report Technical Report 0207. INRIA, (Rocquencourt, France, June) (1997)
11. Torrisi, F.D., Bemporad, A., Bertini, G., Hertach, P., Jost, D., Mignone, D.: Hysdel 2.0.5 - user manual. Technical report, ETH Zurich (2002)
12. Silva, B.I., Richeson, K., Krogh, B., Chutinan, A.: Modeling and verifying hybrid dynamic systems using checkmate. In: ADPM. (2000)