Refinement of Hybrid Systems from Formal Models to Design Languages[*]

Jan Romberg
Systems & Software Engineering, TU München

Christoph Grimm
Technische Informatik, J. W. Goethe-Universität Frankfurt am Main

Abstract

System-level design for discrete-continuous embedded systems is a complex and error-prone task. While existing design languages like SystemC and its extension to the mixed-signal domain, SystemC-AMS, are well supported by a wealth of tools and libraries, they lack both mathematical precision and intuitive, abstract design notations. Graphical design notations with formal foundations such as HyCharts suffer from the lack of tool support and acceptance in the developer community. To overcome the deficiencies of both approaches, we present a design flow from graphical HyCharts to SystemC-AMS designs. This design flow uses a formally founded refinement technique to ensure the overall consistency of the design.

1 Introduction

Embedded control systems are frequently characterized as a mixture of continuous and discrete behaviors. We call such systems discrete-continuous or *hybrid* systems. Because the discrete part of a hybrid system introduces discontinuities in the system's evolution, the behavior of hybrid systems tends to be more difficult to predict and analyze than, for instance, systems of linear differential equations. For the task of high-level design of a control system, it is highly desirable to use representations that accurately reflect both continuous and discrete behavior. In the early stages of a design, this frees the developer from considering implementation details like quantization and sampling, and allows designers to concentrate on the essential features of the functional design.

In our view hybrid formalisms like Hybrid Automata [1], or HyCharts [12] are well suited for precisely capturing the continuous/discrete behavior of a hybrid system. A major advantage of these formalisms is that, being based on a formal semantics, models are susceptible to automated analysis and formal verification. However, as most designs are implemented using digital hardware, there is currently a gap between the capturing and verification of an abstract design in mixed discrete-continuous time, and the discretized design and implementation of such systems. For the later design phases, discrete *approximations* of the hybrid model that explicitly consider quantization and sampling effects are more appropriate.

Hybrid systems combine continuous behavior specified by differential equations with discontinuities introduced by discrete switching logic. With discrete hardware being pervasive in embedded systems, the prevalent way of simulating and implementing hybrid systems is based on discrete-time or discrete-event algorithms. Numerical solvers are one example for such discrete

[*]This work was supported with funds of the Deutsche Forschungsgemeinschaft under reference numbers Br 887/9 and Wa 357/14-2 within the priority program *Design and design methodology of embedded systems*.

algorithms; a variety of variable- and fixed-step algorithms are successfully used for continuous-time simulation, and their capabilities and limitations are well-understood [10]. For implementation or large simulations, simple fixed-step algorithms like Euler forward with quasi-constant performance are widespread.

The effects that introduce deviations to the 'ideal' behavior of a realization (or simulation) can be roughly characterized as:

- Quantization and limitation of the variable's values.
- Quantization of the time. Modeling smooth changes of analog functions would require an infinite number of events/process activations. This is approximated by activation at a finite number of discrete time steps.

It has been recognized by numerous authors [10] that simulation or real-time computation of hybrid system across discontinuities may cause large errors when the design is discretized ad-hoc. For component-level simulation, variable-step algorithms offer good results; however, simulation performance is generally inacceptable for larger (system-level) designs.

SystemC-AMS [5] offers support for mixed solvers for continuous, hybrid, and discrete designs. Cyclic dependencies between different solver kernels are broken into acyclic structures by introducing a sufficiently small delay. The resulting acyclic structure is then ordered in the direction of the signal's flow. After that, the outputs can be computed successively from already know inputs by executing a step in each solver kernel separately. This model of computation is called *static dataflow*, and the delay determines the frequency with which the signals between the signal processing blocks are sampled.

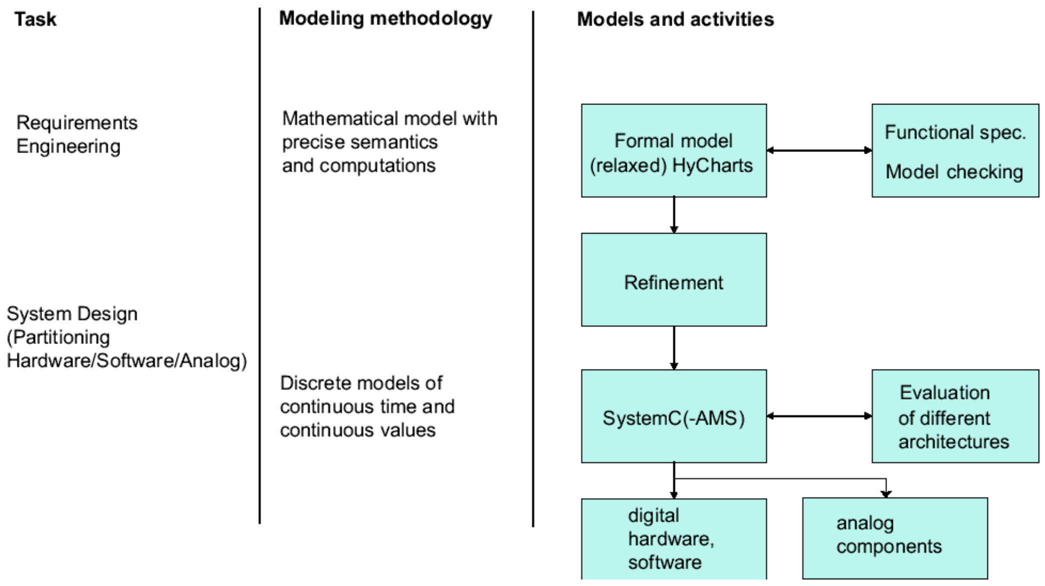
We'd like to employ the approach in the sense that formal systems specifications in HyCharts are translated to fixed-step discrete part in the SystemC-AMS model, and SystemC-AMS's support for mixed-mode simulation is employed for simulating the system along with, for instance, a model of the system's environment.

In this paper, we define a design flow from a formal model of a hybrid system to its discrete realization. We use HyCharts for modeling the hybrid system, and SystemC(-AMS) with static dataflow model of computation for simulation, and as a starting point for hardware/software co-design. Figure 1 gives an overview of the proposed design flow. Starting from a HyChart model, a discretized HyChart is derived and shown to be a time refinement of a relaxed version of the original HyChart. The discrete HyChart can then be mapped to an equivalent discrete-time SystemC model. This model, in turn, is used for simulation and synthesis of both analog and discrete HW/SW components.

Related work Refinement methodologies are known in different domains: In an informal context [6, 4] an executable specification is augmented with further design details in a series of successive design steps. After each step, the design is evaluated by simulation. We call this *design refinement*. E.g. in [6, 5], a SystemC model of a block diagram is refined from a continuous time block diagram to a mixed-signal architecture. The effect of each discretization can be validated by a simulation.

Formally, *behavior refinement* may be defined as the containment of the input/output relation of one (concrete) component in the corresponding relation of another (more abstract) component. [3] describes refinement involving a change of the underlying time model. Much of the work in this paper is based on [12] which discusses time refinement, relaxation, and discretization in the context of HyCharts. A related approach for the hybrid formalism *Charon* is described in [2].

This paper is structured as follows: Section 2 gives an overview of the HyCharts formalism used for specification of hybrid systems, and presents a method for relaxation and time refinement within this framework. Section 3 briefly describes the design and simulation of signal



Task
Modeling methodology
Models and activities
Requirements Engineering
Mathematical model with precise semantics and computations
Formal model (relaxed) HyCharts
Functional spec. Model checking
Refinement
System Design (Partitioning Hardware/Software/Analog)
Discrete models of continuous time and continuous values
SystemC(-AMS)
Evaluation of different architectures
digital hardware, software
analog components

Figure 1: Design flow and translation between HyCharts and SystemC

processing systems using SystemC. Section 4 explains the translation from a discrete HyCharts model to SystemC. In our view, the methodology is not necessarily restricted to the combination HyCharts/SystemC; a similar method may be derived for other formalisms and design frameworks.

2 HyCharts

HyCharts [7] were developed as a graphical notation for hybrid systems, similar in some respect to graphical design notations like UML-RT/ROOM [11], yet formally precise like hybrid automata [1] or related formalisms.

Hierarchical graphs. As a common syntactical representation both HyACharts and HySCharts are based on *hierarchical graphs*. Each hierarchical graph is constructed with the following operators: $\star$ (independent composition of nodes), ; (sequential composition of nodes), $\uparrow$ (Feedback), $\bullet\!\!<$ (identification), $>\!\!\bullet$ (ramification), and $\between$ (transposition). The semantics of both structural and behavioral description is defined by associating a meaning with the graph operators.

Structural description with HyACharts. A HyAChart (Hybrid Architecture Chart) consists of a hierarchical graph whose nodes represent *components*, and whose arcs represent *channels*. The semantics of the hierarchical graph is *multiplicative*: all the nodes in the graph are concurrently active. Being a hierarchical graph, each node, in turn, may have sub-nodes.

Figure 2 shows an example of a HyAChart of an electronic height control system (EHC). The purpose of the EHC, which was taken from an automotive case study, is to control the chassis level of an automobile by means of a pneumatic suspension. A chassis level sH is measured by sensors and filtered to eliminate noise. The filtered value fH is read periodically by CONTROL. CONTROL outputs a target chassis level aH encoding the rate of change due to the operation of two actuators, a compressor and an escape valve.

The basic operations of CONTROL are: (1) if the chassis level is below a certain lower bound, a compressor is used to increase it, (2) if the level is too high, air is blown off by opening an escape valve, (3) whenever the car is going through a curve, the EHC is turned off, (4) whenever

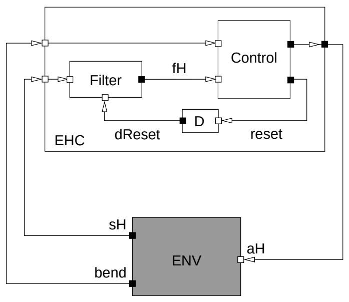


Figure 2: HyAChart of the EHC system

the chassis level is back in the tolerance range, the filter is reset.

For the remainder of this paper, we will concentrate on the Control component as it contains the essential functionality. Extension of our approach to the other system parts is straightforward.

Behavioral description with HySCharts. HySCharts are also based on hierarchical graphs; the graph used in the semantics is obtained from the specification by a simple syntactic transformation. Graphs for HySCharts are interpreted *additively*: Only one node in the graph is active at a given time.

A HySChart defines the (discrete or continuous) behavior of a component. Nodes (rounded boxes) represent control states, and arcs represent transitions. Similar to the Statecharts formalism, HySCharts are hierarchical in the sense that control states may be refined by sub-states.

As an example, Control's behavior is specified by the HySChart shown in figure 3.

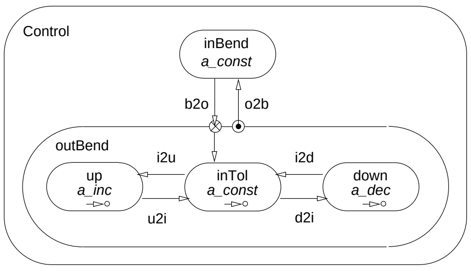


Figure 3: HySChart of the Control component

Control's HySChart has two levels of hierarchy: The topmost state Control has the sub-states inBend (car is driving through a curve), and outBend (otherwise). outBend, in turn, is decomposed into sub-states inTol (chassis level within tolerance), up (raise chassis level), and down (lower chassis level).

Transitions in HySCharts are labeled with *actions*. Actions are a conjunction of a precondition on the component's latched state and current input (*guard*) and a postcondition determining the next data state. We use left-quoted variables $v`$ for current inputs, right-quoted variables $v'$ for the next data state, and plain variables for the latched data state. For variables with discontinuous changes, discontinuities are detected using *timestamps*. We write $v.val`$ for the current value, $v.t`$ for the current timestamp, and $v.t$ for the latched timestamp of v. The special variable *now* refers to a global clock evolving continuously with time.

As an example for an action, *u2i* shown in table 1 expresses that the chassis level must be greater or equal to the lower bound plus some constant ($fH.val` \geq lb + c$), and that a reset event is to be emitted ($reset!$).

States in HyCharts are labeled with *invariants*. Operationally, a transition exiting *can* be taken iff its action guard evaluates to true, and a HySChart *cannot* be in a state whose invariant evaluates to false, therefore "forcing" it to take an outgoing transition. In order to avoid time deadlocks, invariants are required to be *sound*: for each state, there is always either an emerging transition which is enabled, or the invariant evaluates to true, or both.

Control states may be labeled with *activities* specifying the continuous part. Activities describe the continuous evolution of the component's variables when control is in the respective state. Note that the time derivative of a variable v is written as $\dot{v}$.

In our example, activity *a_const* associated with control states INBEND and INTOL specifies that variable aH remains constant.

Note that the input/output relation *Com* resulting from the HySChart is required to be total for all states and inputs. This ensures that a HySChart cannot "reject" certain inputs.

Table 1, middle column, shows the actions, invariants and activities for the HySChart of CONTROL. The ϵ values are introduced by the transformation rules explained in the "Relaxation" paragraph below.

	HySChart	DiSChart
Actions:		
b2o	$bend?$	
o2b	$b2o$	
i2u	$fH.val\text{‘} \leq lb$	
i2d	$fH.val\text{‘} \geq ub$	
u2i	$fH.val\text{‘} \geq lb + c \wedge reset!$	
d2i	$fH.val\text{‘} \leq ub - c \wedge reset!$	
Invariants:		
$inBend_{inv}$	$bend.t = bend.t\text{‘} \vee now - bend.t\text{‘} < \epsilon_{bend}$	$bend.t = bend.t\text{‘} \vee now \bmod T \neq 0$
$inTol_{inv}$	$fH.val\text{‘} \in (lb - \epsilon_{i1}, ub + \epsilon_{i2})$ $\vee\ now - fH.t\text{‘} < \epsilon_{fH}$	$fH.val\text{‘} \in (lb, ub)$ $\vee\ now \bmod T \neq 0$
up_{inv}	$fH.val\text{‘} < lb + c + \epsilon_u$ $\vee\ now - fH.t\text{‘} < \epsilon_{fH}$	$fH.val\text{‘} < lb + c$ $\vee\ now \bmod T \neq 0$
$down_{inv}$	$fH.val\text{‘} > ub - c - \epsilon_d$ $\vee\ now - fH.t\text{‘} < \epsilon_{fH}$	$fH.val\text{‘} > ub - c$ $\vee\ now \bmod T \neq 0$
Activities:		
a_const	$\dot{aH} = 0$ with $\epsilon_{dis}.aH = 0$	$aH' = aH$
a_inc	$\dot{aH} = cp$ with $\epsilon_{dis}.aH = 0$	$aH' = aH + cp{\cdot}T$
a_dec	$\dot{aH} = ev$ with $\epsilon_{dis}.aH = 0$	$aH' = aH + ev{\cdot}T$
Output relaxation constants: $\epsilon_{int}.aH$		

Table 1: Actions, invariants and activities for Control

Semantics. The behavior of a HyChart is specified by the combined behavior of its components. Each of them is formally specified by the *hybrid machine model*. Figure 4, left, shows the machine model of a HySChart. It is constituted of a combinational part ($Com^{\dagger}$), an analog part (*Ana*), a feedback loop with an infinitely small delay (Lim_z), and a projection ($Out^{\dagger}$). The feedback loop, combined with Lim_z, models the state of the machine. At each point in

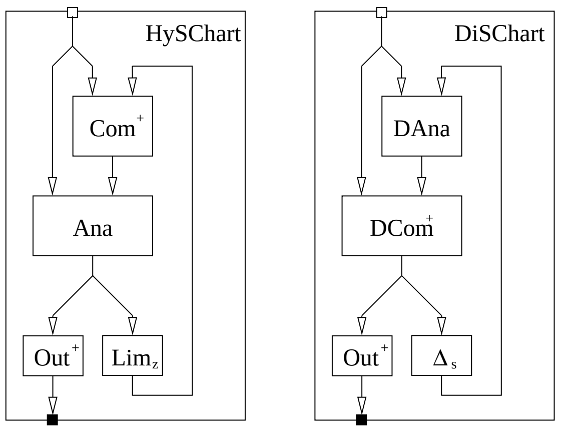


Figure 4: HyAChart of the hybrid machine model

time t, the component can access the received input and the output "exactly before" t. $Com^\dagger$ controls the analog part, and allows discrete manipulations of state variables, but does not have a memory. Depending on the current input and the fed back state, $Com^\dagger$ computes the next state. The analog part uses this next state to select an activity which specifies the continuous flow of the component's variables. The next state may also assert an initial value for the activity. Note that $Com^\dagger$ may alter the component state only for isolated points in time. Between these points in time, $Com^\dagger$ idles. For global composition of HyChart models, HyACharts specify the data flow between components, and HySCharts describe the hybrid machines.

Time Refinement. When discretizing HyCharts, two deviations from the original behavior are introduced: (1) due to sampling, transitions in the discretized model are taken some time after they would have been enabled in the idealized model, (2) sampling in the discretized analog part introduces *discretization errors* and *intersample errors*. Our notion of error refers to the deviation of the discretized signals from the original signals. Discretization errors at sampling instants, while intersample errors are deviations in between samples, assuming that the discretized signal remains constant.

For a tractable design flow, the notion of *behavior refinement* is helpful: A (more concrete) component A is a refinement of a (more abstract) component B if A's input/output relation is contained in B's input/output relation. As it preserves universal safety properties, this kind of refinement is also called *property refinement*. The property refinement considered here is a change of the underlying time model from mixed continous/discrete to discrete. The (possibly) continuous evolution of a value in A is approximated by a series of discrete value changes in B.

Relaxation. While mathematically precise, the semantics of HyCharts or other non-relaxed hybrid formalisms is too "sharp" for discrete-time refinement. *Relaxed HyCharts* accomodate for these uncertainties in the system's behavior: A relaxed HyChart is constructed from a regular HyChart by relaxing its state invariants, analog dynamics, and outputs using a set of transformation rules. The relaxed HyChart therefore allows a larger set of behaviors.

Informally, the transformation proceeds along the following scheme:

Construction of relaxed invariants. In regular HySCharts, *state invariants* are implicitly associated with each control state. The invariant is constructed as the conjunction of the negated guards of outgoing transitions. This results in eager taking of the transitions; we therefore call such invariants *exact invariants*. The relaxation weakens the exact invariants so that some time passes between an action guard becoming true and the corresponding part of the invariant becoming false. We associate a relaxation constant ϵ_x with each discrete and hybrid variable x. The HySChart may remain in the node for ϵ_x time units after the last change of x. For continuous and hybrid values, threshold crossings are relaxed with a similar constant ϵ_a, so transitions must be taken only if the value is significantly above the threshold.

Relaxation of analog dynamics. The relaxation introduces another relaxation constant: For each real valued variable controlled by an activity, $\epsilon_{dis}.v$ specifies the allowed deviation of relaxed from non-relaxed behavior. For refinement, this relaxation accomodates for the discretization error.

Relaxation of output variables. The output relaxation is introduced by inserting an additional relaxation component at the HySChart's output interface. The relaxation component ensures that for each output variable x, all behaviors with a maximum deviation $\epsilon_{int}.x$ are included in the HySChart's behaviors. Note that the relaxation may also introduce discontinuous jumps in the evolution of x. In our refinement technique, this relaxation accomodates for the intersample error.

Table 1, middle column, already shows the relaxed invariants and activities. A more detailed and formal treatment of the transformation rules can be found in [12].

Discrete HyCharts. *Discrete HyCharts* were developed for describing the discrete-time refinements of HyCharts models. Both regular and discrete HyCharts use HyACharts for structural description; for behavioral description, *DiSCharts* are introduced. Like HySCharts, DiSCharts are hierarchical, sequential control-flow graphs, but in contrast to HySCharts the underlying time model is discrete only. The DiSCharts machine model is similar to HySCharts (Figure 4, right): the infinitely small delay Lim_z is replaced by a unit delay Δ_s, and the order of Ana and Com is reversed so that Com can immediately react to state changes caused by Ana.

Construction of the DiSChart from HySChart. Table 1, right column, shows the invariants and activities for the DiSChart of Control (as the control flow is generally not modified when going from HySCharts to DiSCharts, the graphical representation is equivalent). T s the fixed sampling period for the discrete HyChart. The DiSChart was derived using a discretization method from [12]. In principle, other (fixed-step) discretization methods are possible.

The method makes the following assumptions on the chart's inputs and variables: (1) All variables which change continuously with time must be Lipschitz constrained with constant l, that is, there exists an l such that the time derivative is within $\pm l$.For the Control example, we assume a Lipschitz constant fl for variable fH. (2) For all continuously changing variables, a maximum error e is given. In the example, fH has a maximum error of fe. (3) For each hybrid and discrete input channel, a minimum event separation m has to be provided.

It can then be shown [12] that the derived discrete HyChart of the EHC system is a discrete-time refinement of the relaxed HyChart if the sampling period T is chosen such that the following inequality holds:

$$T \leq \min\{\epsilon_{bend}, \frac{\epsilon_{i1} - 2 \cdot fe}{fl}, \epsilon_{fH}, \frac{\epsilon_{int}.aH}{cp}, \frac{\epsilon_{int}.aH}{|ev|}\}$$

With the discretized HyCharts model, we are now ready to introduce SystemC-AMS as the target format for the translation in the next section.

3 Modeling Hybrid Control Systems with SystemC

For the realization of hybrid control systems, continuous functions are realized by discrete hardware/software systems. The design of such systems starts with an executable specification, e.g.

in Matlab/Simulink or SystemC. SystemC is a C++ library which supports the executable specification and design of complex, embedded hardware/software systems. In the following, we give a brief overview of means to model discrete systems in SystemC, and methods to model signal processing systems in SystemC(-AMS).

Modeling and Simulation of Components In SystemC a component is an object of the class `sc_module`. Discrete behavior of components is specified by methods or threads. For example, we can model the behavior of a PI controller by a process as follows, provided the signal `clk` activates the process at constant time steps:

```
double state, k_i, k_p;
class pi_controller: public sc_module
{
   sc_in<double> input; // in port
   sc_in<bool> clk;     // clock
   void do_step { state += k_i*input.read();
                  output.write(state+k_p*input.read());};

   SC_CTOR(pi_controller) {
     SC_METHOD(do_step) sensitive << clk;
     // executes do_step at each event on clk
   }
}
```

Of course, the continuous behavior of analog components such as the above integrator is only approximated. In the same way as above, one can model components used in section 4, such as unit delay, multiplexer or arithmetic functions.

Modeling and Simulation of Block Diagrams If a number of blocks with discrete or continuous behavior, that communicate is combined to an architecture we must introduce a model for communication and synchronization. In SystemC, the communication between processes respectively modules is modeled by channels (signals). For example, we can specify a structure, where the controller gets an input from a system `s1` as follows:

```
sc_signal<double> a, clk; // instanciate channels
pi_controller ctrl1();    // instanciate pi controller module
system s1();              // yet another module
ctrl1.input(a); ctrl1.clk(clk); // connect modules via channels
s1.output(a); s1.clk(clk);
```

In the above example the method `ctrl1.input(a)` is called *before* simulation starts. The method notifies the simulation manager about a connection between an abstract port (which is only an interface) and a signal, which realizes this interface.

For system level design of signal processing systems the static dataflow model of computation as introduced in section 1 is more appropriate that discrete event simulation. It is actually implemented in prototypes of SystemC-AMS ([15]).

In SystemC-AMS clusters of signal processing blocks are controlled by a cluster manager or coordinator as shown in figure 5. The coordinator determines a schedule of the modules in the data flow's direction using the information of ports and directed signals. Then the coordinator simulates all blocks in constant time steps in this order. For example, in figure 5, for a given input of the SystemC kernel and a known delayed value, the coordinator would first ask block *sigproc1* to compute his outputs, then *sigproc2*, and finally *sigproc3*. After a delay the same procedure would start again, and so on.

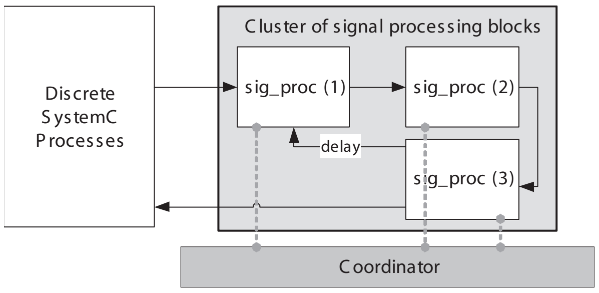

Discrete SystemC Processes
Cluster of signal processing blocks
sig_proc (1)
sig_proc (2)
sig_proc (3)
delay
Coordinator

Figure 5: SystemC-AMS simulation of a signal processing cluster

In the following, we assume a realization in SystemC-AMS. For visualization, we use the Simulink block diagram editor.

4 Translation of discrete HyCharts to SystemC

In this section we show how a discrete HyChart can be translated into an equivalent composition of SystemC blocks. For the current lack of a suitable block diagram tool in the SystemC context, we chose Simulink as a block diagram editor in order to demonstrate the translation. We assume some kind standard block library exists with a set of *arithmetic*, *boolean comparison*, *unit delay*, and *signal routing* blocks. Signal routing includes *multiplex* and *demultiplex*, *switch* (given some n and a vector $\vec{v}$ as input, output the n-th component of $\vec{v}$), and *selector* (given some vector $\vec{v}$ as input, output a vector $\vec{v'}$ with some of $\vec{v}$'s components) blocks. We furthermore assume that simple expressions as in the discrete HyChart's actions and activities are directly encoded as SystemC-AMS blocks.

The discrete HyChart has to meet the following criteria in order to be translatable: (1) it is deterministic, i.e. its internal state and its outputs are completely determined given an input, (2) there are no delayless loops in the component network, (3) the identification connector $\bullet\!\!<$ is not used in HyACharts (identification has not been used in any HyChart model so far). (4) it is total on all inputs and states, hence inputs cannot be rejected. The latter condition is always fulfilled by our refinement procedure, as totality of the discrete *Com* part's input/output relation follows directly from totality of the original (non-relaxed) HyChart's *Com* part. Each HyChart operates on a data state $\sigma \in \mathcal{S}$, an input $\iota \in \mathcal{I}$, and a control state $\kappa \in \mathbb{N}$. $\mathcal{S}$ is composed of the product of the types of all variables accessed in the HyChart's activities. The control state κ is some natural-number encoding of the HyChart's control states. Extension of the encoding to hierarchical states by shifting namespaces is described in [13].

Translation of multiplicative and additive graphs. As a useful prerequisite for the translation, the multiplicative (parallel) composition operators in HyACharts naturally reflect composition in dataflow-oriented design languages like Simulink or SCADE/Lustre [8]. We therefore do not show an explicit example for HyAChart translation, and concentrate on HySCharts instead. The additive (sequential) composition of m nodes in HySCharts is generally translated as m parallel blocks, with a switch block choosing the appropriate output according to the current control state. We'll see examples for the translation below.

Machine model. The top-level translation of a HySChart follows directly from the discrete-time hybrid machine model sketched in Figure 4, right: The discrete-time variation of the analog

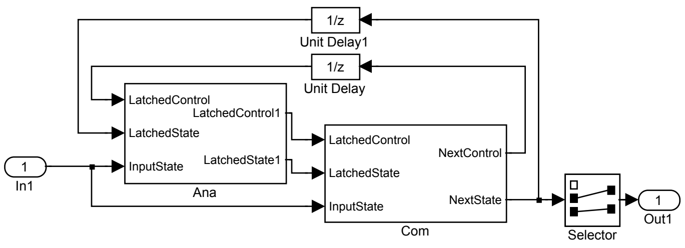


Figure 6: Block diagram translation for *Control*

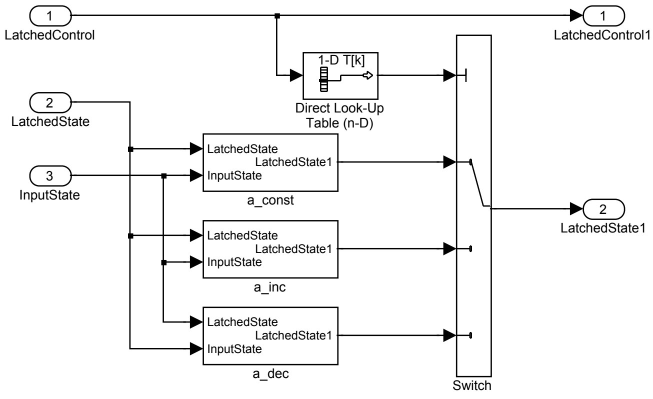


Figure 7: Block diagram translation for *Ana*

part, *Ana*, computes the next state $(\sigma.fH(kT), \sigma.reset(kT), \sigma.bend(kT))$ from the current inputs $(\iota.aH(kT), \iota.bend(kT))$, the latched control state $\kappa((k-1)T)$, and the latched data state $(\sigma.fH((k-1)T), \sigma.reset((k-1)T), \sigma.bend((k-1)T))$. The combinatorial part *Com* then computes the next control and data state from the current input and *Ana*'s outputs. The chart's output $o \in \mathcal{O}$ is computed by projecting the data state onto $\mathcal{O}$ using a selector block (Figure 6). The unit delay in the feedback loop, finally, is realized with unit delay $(\frac{1}{z})$ blocks.

Analog part. Specification of the discrete evolution law is typically done with difference equations, such as the activities in table 1. In the translation, each single activity in the discrete HyChart corresponds to a SystemC-AMS block computing the corresponding difference equation. The currently active activity is then selected by an m-ary switch block based on the current control state. Note that the control state is not controlled by *Ana* and simply fed through. Figure 7 illustrates the translation.

Combinational part. In HyCharts, the semantics of the combinational part is given in terms of a syntactical transformation of primitive control states (additive *nodes*) to hierarchic graphs. Each node of the hierarchic graph is entered through one of its m entry points, executing their entry actions, or through an explicit wait entry point if there is no change in control state. The appropriate action is chosen by evaluating the *guards* associated with each action. Hierarchical control states are transformed in a similar manner. In our translation, the m action guards are translated to m parallel SystemC-AMS blocks which compute the guards. The next control state is then chosen using, for instance, an m-ary *look-up table block*; determinism of the chart ensures that only one guard at a time evaluates to true. If none of the guards evaluates to true, the chart remains in the same control state, corresponding to the invariant guard in the hierarchic graph. Figure 8 shows the translation for the EHC system's *Com* block. Note that the control state hierarchy has been flattened so that inBend, up, inTol and down are on the same level of hierarchy.

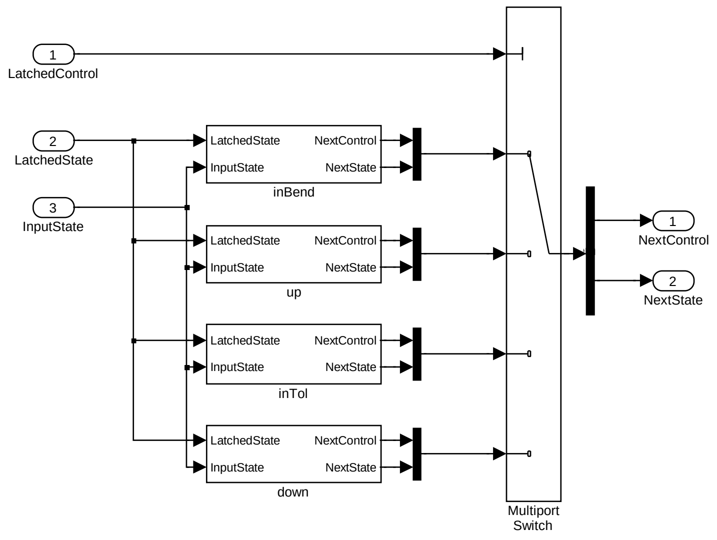


Figure 8: Block diagram translation for *Com*

Actions and activities. In this translation, evaluation of assignments, arithmetic expressions, and boolean predicates in actions and activities is handled at the block level. In SystemC-AMS, such expressions are simply encoded as equivalent C language statements within blocks. Note that in the C language translation, the term $now \bmod T \neq 0$ is removed from each of the DiSChart's invariants as the simulator enforces discrete steps anyway.

As mentioned above, extension of the above translation to parallel composition of discrete HyCharts is straightforward as multiplicative composition in HyACharts is naturally represented in block diagrams.

5 Conclusion and Future Work

Conclusion. We have presented an integrated approach for the design and synthesis of hybrid systems using a formal, graphical design notation to capture abstract designs of hybrid systems. Based on a relaxed version of the model, the design is discretized and translated to a model in a design language. Correctness of the translation is ensured by evaluating a set of constraints, ensuring that the discretized model is a time refinement of the relaxed model.

Future Work. In principle, the above approach may be extended towards other discrete-time languages used for synthesis of HW/SW systems. Synchronous languages like Lustre [8] or AutoFocus[9] would be natural targets for our translation. As yet, HyCharts are not directly supported by design tools. More tool support for editing and for applying transformation rules, for instance in the MaSiEd environment [14], would clearly be desirable at this point.

Acknowledgements. Thanks to Thomas Stauner for commenting on a draft version of this manuscript.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[2] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement of hierarchical hybrid systems. In *Proceedings of HSCC 2001*, LNCS 2034. SpringerVerlag, 2001.

[3] M. Broy. Refinement of time. In Th. Rus M. Bertran, editor, *Transformation-Based Reactive System Development. ARTS'97*, Lecture Notes in Computer Science 1231, pages 44–63, 1997.

[4] D. Gajski, F. Vahid, and S. Narayan. A System-Design Methodology: Executable-Specification Refinement. In *The European Design Automation Conference (EURO-DAC'94)*, Paris, France, February 1994. IEEE Computer Society Press.

[5] Ch. Grimm. Modeling and Refinement of Mixed Signal Systems with SystemC. In *Methodologies and Applications*, Boston/London/Dordrecht, 2003. Kluwer Academic Publishers.

[6] Ch. Grimm, W. Heupke, Ch. Meise, and K. Waldschmidt. Refinement of Mixed-Signal Systems with SystemC. In *Design and Test in Europe 2003 (DATE '03)*, Munich, Germany, 2003.

[7] R. Grosu, T. Stauner, and M. Broy. A modular visual model for hybrid systems. In *Proc. of FTRTFT'98*, LNCS 1486. Springer-Verlag, 1998.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[9] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[10] P.J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *Proceedings of HSCC 1999*, LNCS 2034, pages 165–177. SpringerVerlag, 1999.

[11] B. Selic, G. Gullekson, and T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, New York, 1994.

[12] T. Stauner. *Systematic development of hybrid systems.* PhD thesis, Technische Universität München, 2001.

[13] T. Stauner and C. Grimm. Übersetzung von HyCharts in HDFG. In *ITG/GI/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Berlin, Germany, 2000. VDE-Verlag.

[14] T. Stauner, A. Pretschner, and I. Péter. Approaching a discrete-continuous uml: Tool support and formalization. In *Proc. UML'2001 workshop on Practical UML-Based Rigorous Development Methods*, pages 242–257, Toronto, October 2001.

[15] A. Vachoux, Ch. Grimm, and K. Einwich. SystemC-AMS Requirements, Design Objectives and Rationale. In *Proceedings of Design, Automation and Test in Europe (DATE'03)*, 2003.