HEAVEN: A Framework for the Refinement of Heterogeneous Systems*

Rüdiger Schroll, Christoph Grimm, Klaus Waldschmidt

Abstract

A key issue in system design is the evaluation of different architectures. For this task, designers create models of these architectures. In industrial practice, designers create such models by modifying and combining existing models. This leads to systems that combine different models of computation and different kinds of signals at different levels of abstraction. Especially in signal processing systems, very different kinds of signals are combined.

Without further actions, such as the programming of converters or adapters, the resulting models are inconsistent and cannot be simulated directly. In the following, we discuss the use of polymorphism for automatically converting inconsistent models into consistent ones. We introduce polymorphic signals, and give an overview of a prototypical implementation HEAVEN with an example that demonstrates the use of polymorphic signals.

1 Introduction

A key issue of system design is the analysis of different architectures. Especially for signal processing systems, the 'design space' is often huge. Design issues such as partitioning (analog, digital ASIC, DSP+Software), determination of sample frequencies, bit widths, or precision of analog components determine quality, performance and costs of the system. In order to analyze and to verify the behavior of different architectures, a model of each architecture is created and simulated.

Modeling languages such as VHDL-AMS allow designers the modeling and simulation of mixed-signal systems at different levels of abstraction. However, for modeling an architecture with another partitioning, or with different sample rates, significant modifications of the models are required. Most notably, these modifications are due to the fact that VHDL(-AMS) requires a concrete specification of how communication and synchronization are implemented. Therefore, at the interfaces, modeling often requires nearly the same effort as a full design.

In *model based design*, designers use different platforms with pre-defined schemes for communication and synchronization (models of computation, MoC). Different MoCs can be chosen, depending on the architecture to be modeled: Analog circuits can be modeled by equations, DSP methods for example by static dataflow with constant time steps. Because the MoC specifies the behavior of communication, there is no need for the designer to model communication in an explicit way. Therefore, model based design allows designers evaluation of different architectures in an earlier stage of the design process. Particular research on the use of different MoCs and model based design has been done within the design framework Ptolemy [1].

The aim of *model refinement* is a stepwise, successive design and verification process, where each design issue can be evaluated immediately. This means, that simulation of 'incomplete' designs

*This work has (in part) been supported within the BMBF/edacentrum project 'SAMS'.

should be possible, and that the effort for changing a model must be small. Compared with model based design, this permits an earlier evaluation of system properties, can increase re-usability of models, and provides a direct link between a design issue and its problems.

In order to demonstrate new methods for model refinement, we use the (prototypical) design framework HEAVEN (**HE**terogeneous Systems Refinement, **A**nalysis and **V**erification **EN**vironment) shown by figure 1. HEAVEN supports the refinement of signal processing systems to different architectures using SystemC-AMS[2]. We consider two kinds of refinement: The *refinement of computation* introduces properties such as sample frequencies, bit widths, precision, etc. by choosing a model of computation and signal types. In the *refinement of interfaces*, the behavior implicitly introduced by choosing a model of computation is made explicit by defining a physical implementation thereof, e.g. using clock and enable signals.

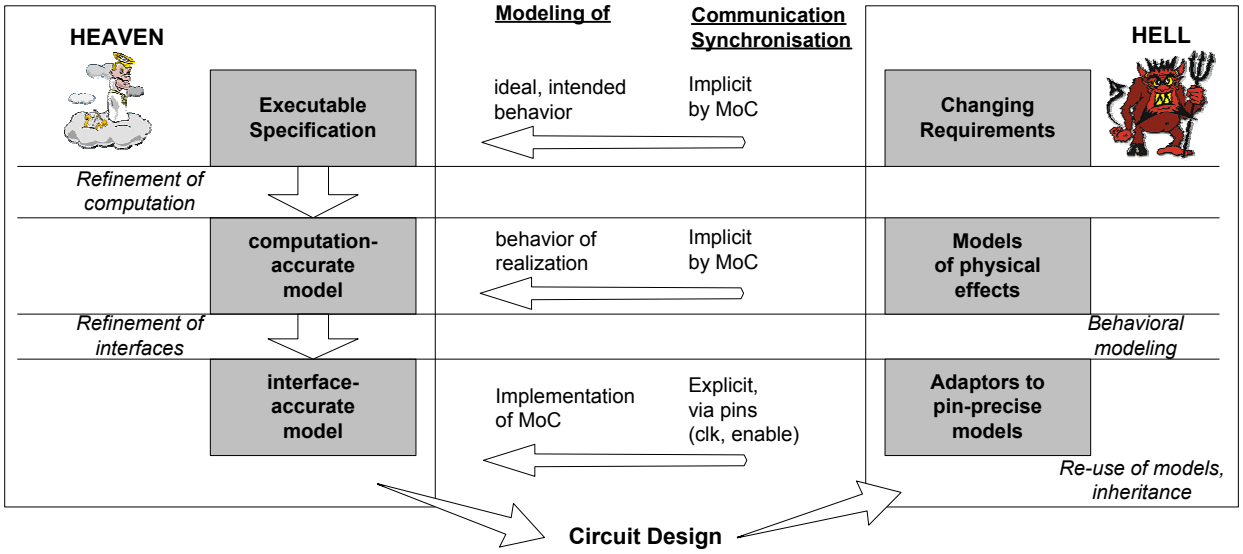

Figure 1: Refinement of signal processing applications to mixed-signal circuits with HEAVEN/HELL.

The result of the model refinement is a fully specified architecture. The design of signal processing systems without considering physical effects and non-ideal behavior from analog or digital implementations is not realistic. Therefore, HEAVEN is supported by HELL (**HE**terogeneous Systems mode**l**ing **L**ibrary; figure 1 right). HELL provides behavioral models of physical effects that can be added to the 'ideal' behavior assumed in HEAVEN.

In this paper, we give an overview of HEAVEN's features that support model refinement. Most notably, model refinement is supported by polymorphic signals for signal processing systems, and by adapter classes that implement a model of computation by 'pin-precise' models. Polymorphic signals implicitly convert signal types that connect MoCs used in the design of signal processing systems. Therefore, polymorphic signals allow designers to compose models in an intuitive and interactive way, just by connecting existing blocks.

Related work The use of polymorphism for modeling heterogeneous systems is not new. Basic ideas for polymorphic models and signals in HEAVEN are also properties of hybrid data-flow graphs [3, 4], where signal types are converted implicitly, and the semantics of nodes is defined by firing rules. An implicit conversion of different signal types is also provided by Matlab/Simulink.

However, Matlab/Simulink is restricted to block diagrams with discrete or continuous signals, and does not support modeling of analog or digital circuits or software.

SystemC 2.0 introduces a very generic approach, where signals are accessed via interfaces which can be realized in different ways. This allows one to introduce different models of computation by using different implementations of the interface [5]. However, the type checking between the interfaces is very strict, and in order to combine different models of computation, one has to use converter modules, for example.

In Ptolemy II/Chess [1, 6], behavioral types provide basically the same functionality as the signal interfaces in SystemC 2.0. In extension to SystemC 2.0, interface automata permit the coupling of different models of computation provided there is a subtype relation between them[6]. Note that the subtype relation applies to the value types, and to the protocols that implement a model of computation ('behavioral types'). Unfortunately, a subtype relation often does not exist, or can even be misleading because semantic is not considered.

In [7], we introduce polymorphic signals for signal processing systems. Polymorphic signals provide methods that translate communication in different MoCs and at different levels of abstraction. However, the implementation described in [7] is restricted to discrete event (DE) and static dataflow (SDF) MoCs. In the following we introduce application specific semantic types and a polymorphic signal class, that cover the MoCs used in signal processing applications at different levels of abstraction. Compared to Ptolemy, polymorphic signals are application specific, and are not a generic modeling property. The restriction to a domain of applications such as signal processing applications has the advantage, that we can assume that all (polymorphic) signals are approximations of an 'ideal', continuous-time signal. This gives all conversions an intuitive understanding.

Section 2 gives a rough overview of SystemC-AMS, describes the refinement methodology, and introduces general requirements of polymorphic signals for the refinement of signal processing systems. Section 3 describes a polymorphic signal class for the modeling of signal processing systems. Section 4 describes the application of polymorphic signals in a case study.

2 Refinement with HEAVEN

HEAVEN is built on top of SystemC and SystemC-AMS [8, 2]. SystemC-AMS, resp. an early prototype, the ASC-Library [7], permits the modeling and simulation of signal processing systems. In the following we first give a brief description of SystemC-AMS. Then, we describe the refinement of signal processing applications to different mixed-signal architectures, and motivate polymorphic signals which support such a refinement.

2.1 SystemC-AMS

Layered approach In SystemC 2.0 systems are specified by a structure of *modules*. The modules are connected by directed *signals*. Modules access signals via an interface, which is accessed via *ports*. SystemC-AMS provides means for the modeling of signal processing systems in SystemC. SystemC-AMS extensions are structured in three layers[8]:

The *view layer* allows the designer the specification of behavior in different models of computation such as transfer functions, netlists, or a cluster of signal processing functions in the static dataflow MoC.

The *solver (simulator) layer* provides means which execute a specification given at the view layer, e.g. a coordinator which implements the static dataflow MoC, or which solves linear and

non-linear differential equations.

The *synchronization layer* couples different solvers (simulators). For coupling different simulators, the static/synchronous data-flow (SDF) model of computation is used. Note, that both synchronization layer and solver layer introduce an underlying model of computation, but with different aims and requirements: The synchronization layer couples simulators which might also be external simulators such as SPICE. The solver (simulator) layer provides different means for the modeling and simulation of signal processing systems in SystemC.

Coordinator-Interface Instances at the view layer have a unique interface (coordinator-interface). This interface allows the coordinator to control the execution of these objects, as well as their communication and synchronization.

Figure 2 gives an overview of a SystemC-AMS model which consists of discrete-event processes (left), and a cluster in SDF model of computation (modules 1-3, right). Before simulation starts, the SDF coordinator schedules the blocks of the SDF cluster. During simulation, the SDF coordinator executes the modules for each time step. In order to control execution of each module, the coordinator has access to all modules via the coordinator interface. Different simulators (here: SDF coordinator, and SystemC 2.0) are coupled via static data-flow MoC at the synchronization layer.

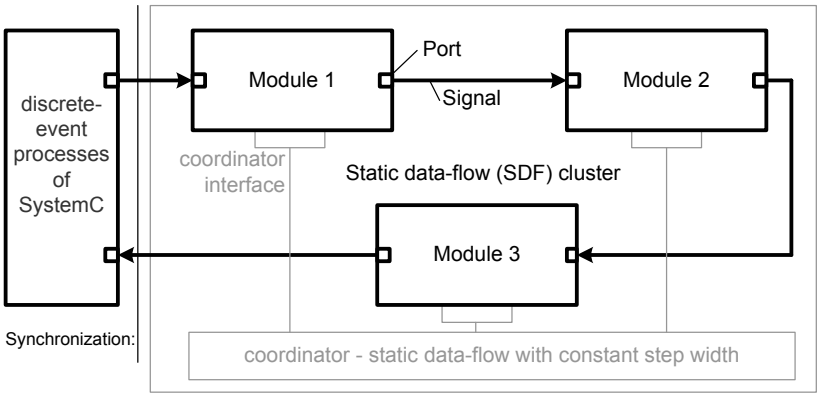

Figure 2: A model in SystemC-AMS.

2.2 Refinement of Computation in Signal Processing Systems

Executable specification The design of signal processing systems begins with a block diagram which describes basic principles of the system. Sample frequencies, range of values, and bit widths are not yet known. For the first simulations, designers use the MoC 'continuous-time (CT) signal flow'. This model of computation assumes that all connections between modules have the semantics of mathematical equations, and that there is no order of execution or width of time steps that comes with this model of computation. Furthermore, signals have no limitation, and no quantization. Table 1 gives an overview of the modeling properties used in the executable specification.

Note, that for simulation of the CT signal-flow model of computation, a discrete algorithm is required. This algorithm solves the mathematical equations, and determines the width of time steps. The time steps introduce an error. This error can be reduced by reducing the time steps, depending on the estimated error. Then, all blocks are simulated in the signal flow's direction. Cyclic dependencies can be broken by insertion of a delay. This is basically the static dataflow MoC, where the execution of the blocks is controlled by the estimated error.

Executable specification		
Signal	value type	No limitation
		No quantization
	sampling	No sampling
MoC	Continuous-time signal flow	

Table 1: Executable specification: model of computation and signals.

Computation-accurate model One aspect of system design is the evaluation of deviations introduced by different realizations. For signal processing systems, there are realizations with fundamentally different behavior: digital signal processing using a DSP or an ASIC, and analog realization. We can easily model the behavior of these implementations by replacing the MoC and/or the signal types of the executable specification by more appropriate ones, which implicitly include properties of a realization.

Properties of digital signal processing systems that have to be evaluated are sampling frequencies f_s, quantization steps Q, and range of values (limitation). A useful MoC for modeling digital signal processing is static dataflow with constant time steps $1/f_s$. Appropriate signal values are integers that model Q and the realization's range of values.

Properties of analog circuits that are modeled are limitation, limited band width and precision. Table 1 gives an overview of the modeling properties used in a computation accurate model.

DSP behavior		
Signal type	value type	Limitation $[lb, ub]$
		Quantization Q
	time steps	t_s
MoC	Static dataflow with constant time steps t_s	
Analog behavior		
Signal type	value type	Limitation $[lb, ub]$
		Precision S/N
	(time steps)	min. time step $t_{s,min}$
MoC	Continuous-time signal-flow (simulated by static dataflow with adaptive time steps)	

Table 2: Analog and DSP behavioral model: models of computations and signal types

The refinement of computation successively replaces modules of the executable specification by modules that model the implementation. The modules that model the implementation use a maybe different model of computation and a different signal type in order to model properties of the implementation. This especially affects the interaction with other blocks via ports, and may introduce incompatibilities or inconsistencies. Potential changes affect

Value types: Range and quantization/precision are restricted, e.g. from general 'real' to an 'integer' representation with limitation.

Interfaces: Changing the model of computation requires use of other interfaces and different protocols for the transport of data.

Semantics: Data is not only transported, but also might have different meanings, e.g. a sequence of bank account numbers, or an approximation of an continuous-time signal, or even nodes with Kirchhoff laws.

Because of these changes, the intuitive composition of a new model by just replacing a module by a more detailed one will not work: In most cases the resulting models are inconsistent, and require

further actions to convert signal types, protocols, and semantics. Figure 3 gives an example for such inconsistencies due to refinements: The left two blocks have been replaced by models of a DSP implementation, and the right block models the continuous-time environment. Furthermore, the value type 'real' has been replaced by 8-bit numbers modeling the range of values from 0 to 255 ($Q = 1$) with limitation. Note, that the semantics of the signals remains unchanged: the signals are approximations of a continuous-time and continuous-value signal.

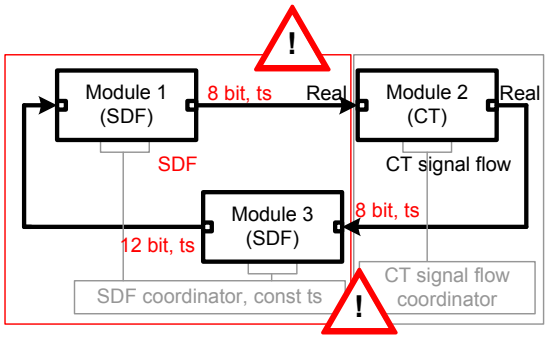

Figure 3: Computation-accurate model with inconsistencies due to refinement steps.

Polymorphic signals Of course, designers can manually write converters that adapt value types, interfaces and consider changing semantics. In a limited range, this can be done automatically considering that range of values (and interfaces) are compatible with subtypes. Behavioral types in Ptolemy II convert protocols by construction of a common automata, but without considering the meaning of signals. This extends compatibility, but does not consider semantics of the data transported via a signal. Semantic issues can only be treated in an implicit or automatic way, if we know the semantics. In order to allow us to convert signals in a more general way, we assume that signals have *semantic types*. A semantic type of a signal is an abstract interpretation of a signal. In the following, we consider signal processing applications. The knowledge of the semantics allows us to formulate different views of one signal, that depends on the signature (interface, value type) used to access the signal. We call such signals *polymorphic signals* (for a domain of applications).

2.3 Refinement of Interfaces

A second aspect of system design is the explicit realization of the communication/synchronization which is implicitly specified by a model of computation. This can be done by the refinement of interfaces. The refinement of interfaces transforms the computation-accurate model to a model that has all ports of the implementation.

In digital ASICs, communication is realized by a clock signal and a controller that uses enable signals to control the communication and synchronization of the single modules. For specification of the ASIC itself at the register transfer level, the discrete event model of computation is used. Note, that there are already approaches for the refinement of communication, such as SystemC^{SV}, and the Master-Slave Library. However, they do not consider the fact that (at least in the ASC library and SystemC-AMS) the execution of the modules is controlled via a coordinator interface.

Figure 4 shows the refinement of interfaces. A module inherits an 'adapter class'. The adapter class translates clock and enable signals to an activation of the module via its coordinator interface.

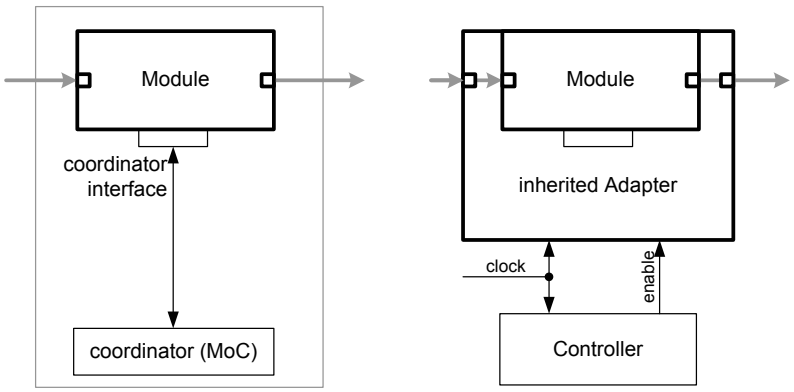

Figure 4: Refinement of interfaces by an inherited adapter class that uses the coordinator interface.

3 Polymorphic signals for signal processing applications

In the following, we describe a polymorphic signal for signal processing applications. In signal processing applications, signals are more or less good approximations of continuous-time signals. Such systems are specified with the following models of computation, depending on the level of abstraction, and the implementation:

- Continuous-time signal flow (simulated by static dataflow with adaptive time steps).
- Static dataflow with constant time steps, but very often with different data rates resp. time steps (multi-rate systems).
- Discrete event system.
- Netlists.

As motivated in section 2, the refinement of signal processing systems is characterized by modules with different sample rates, ranges of values, etc. The polymorphic signal for signal processing applications supports this refinement by providing the following functionality:

- It implicitely adapts the range of values: The range of values of the writing port is adapted to the range of values of the reading port.
- It implicitely converts sample rates: The signal can have different samples rates for writing and reading ports.
- It implicitely converts different models of computation: The polymorphic signal can be read or written from the above mentioned models of computation.
- The polymorphic signal provides means for specification (or modeling) of noise and deviations for semi-symbolic analysis in HELL [9].

Polymorphic signals can actually be used to couple modules in the supported models of computation without requiring the insertion of additional converters. In case that analog netlists are coupled, the polymorphic channel might even hide the complexity of simulator coupling — a designer just sees the channel in SystemC-AMS, and an additional node in the analog simulator.

Implementation In SystemC, signals are accessed via ports with an interface that specifies a set of methods. Modules call these methods. A signal that is connected to a port must provide concrete methods that implement the methods called from the modules via the ports.

For each model of computation a port class is provided, for example `asc_sdf_const_in` for static dataflow with constant time steps. At the ports of the modules, attributes are specified that give additional information about the semantic interpretation of the signal to be accessed, such as:

- `value_unit` and `value_size` can specify a physical size that is associated with the abstract value at port.
- Boundaries for the range of values $[lb, ub]$.
- Time steps and rates of static (multi-rate) data flow.
- `max_deviation` and `max_noise` can specify allowed deviations of signals (for use in HELL, [9]).

If there are inequalities or incompatibilities between the ports that access a polymorphic signal, a conversion has to occur in order to permit a simulation. By default, virtual methods are called. These methods give a warning, and call simple conversion methods. The conversion methods can be overloaded by more appropriate ones, if needed.

`value_unit` and `value_size` of the reading port are compared with the writing port.

The range of values is checked and converted as follows: Let lb_{write} and ub_{write} be the lower and upper bounds of the writing port and lb_{read} and ub_{read} the lower and upper bound of the reading port. If the bounds are not equal, there might be a problem in the design; therefore, a warning is given. Then, by default, the polymorphic signal maps a written value $v_i \in [lb_{write}, ub_{write}]$ from the range of values of the writing port to a value $v_{i,read}$ from the range of values of the reading port $[lb_{read}, ub_{read}]$ as follows:

$$v_{i,read} = v_i * mult - lb_{write} * mult + lb_{read} \text{ with: } mult = \frac{(ub_{read} - lb_{read})}{(ub_{write} - lb_{write})}$$

The polymorphic signal can be written/read from ports of different models of computation. We use the following approach: The polymorphic signal inherits and implements the interfaces of all port types that are compatible with the signal as shown in figure 5. The methods that implement the interfaces translate read- or write- accesses into an internal, abstract representation.

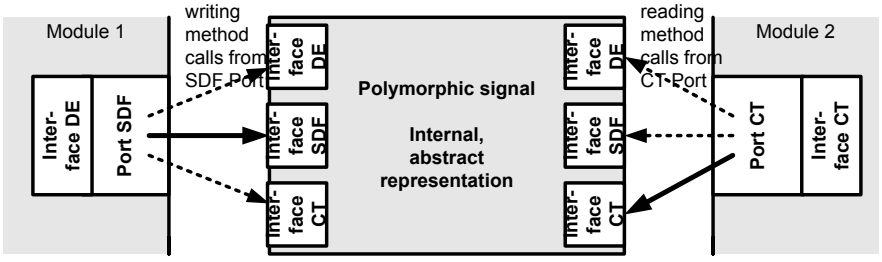

Figure 5: Implementation of polymorphic signals in SystemC-AMS.

In the internal, abstract representation, signals are represented by a queue of tuples $(value, time)$ ('events', 'samples'). An event (v_i, t_i) describes the point of time t_i in sc_time and v_i is the into

double converted value that was written at point of time t_i. (v_i, t_i) is the oldest element in the queue, and (v_j, t_j) $((v_{i+5}, t_{i+5}))$ is the newest element in the queue. The value is of the type double, because all value types are subtypes of double. The point of time is of the type `sc_time`, which is a long integer, which allows us representation of all possible points of time.

The size of the queue n is bounded and determined as follows:

$$n = \frac{t_{s,max} * k}{t_{s,min}}$$

where $t_{s,max}$ is the maximum possible time step, k the factor of multi-rate dataflow, and $t_{s,min}$ is the smallest possible time step.

If there are more than n events in the queue the oldest event leaves the queue. All incoming values will be saved that way, independent which model of computation is used. Then, the queue of n events $(value, time)$ describes the development of a signal in a time frame that covers at least the $t_{s,max}$ of th

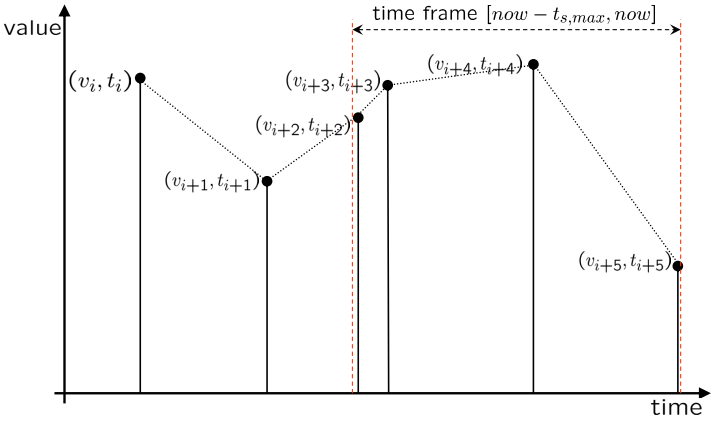

Figure 6: Abstract representation in polymorphic signal class by a queue of events $(value, time)$.

In general, a queue of a signal is given by:

$$(v_i, t_i), (v_{i+1}, t_{i+1}), (v_{i+2}, t_{i+2}), ..., (v_j, t_j)$$

Figure 6 shows an example of a queue

$$(v_i, t_i), (v_{i+1}, t_{i+1}), (v_{i+2}, t_{i+2}), (v_{i+3}, t_{i+3}), (v_{i+4}, t_{i+4}), (v_{i+5}, t_{i+5})$$

with $j \geq i$, $j - i + 1 \leq n$ and n is the maximum buffer size.

If $j - i + 1 = n$ (n = 6, the queue is full) and a new element (v_{j+1}, t_{j+1}) $((v_{i+6}, t_{i+6}))$ is written, the oldest element is removed. The queue becomes

$$(v_{i+1}, t_{i+1}), (v_{i+2}, t_{i+2}), ..., (v_j, t_j), (v_{j+1}, t_{j+1})$$

and in the example

$$(v_{i+1}, t_{i+1}), (v_{i+2}, t_{i+2}), (v_{i+3}, t_{i+3}), (v_{i+4}, t_{i+4}), (v_{i+5}, t_{i+5}), , (v_{i+6}, t_{i+6}).$$

In the following, we describe the methods that modify (read/write) the abstract representation of signals. Synchronization and simulator coupling are implemented in SystemC-AMS resp. the ASC

library (see [7, 2]). This synchronization first executes the modules of the AMS extensions using the last values from the discrete event simulation, and then executes the discrete event simulator.

The following methods are used for writing the queue; the size of the queue is limited, and adding a newer element automatically removes the oldest element:

Writing from SDF port: An event is added to the queue. For multi rate dataflow, several events can be added at once.

Writing from DE port: The value of the event to be written is compared with the last event's value. If the values are different, `request_update()` is called and triggers an event in the DE model. If the times are equal, the last event is deleted. Finally, the new event is added to the queue. This ensures that writing DE processes only add one event (t_i, v_i) to the queue.

Writing from CT port: An event is added to the queue. This method is applicable to CT signal flow model of computation. For netlists, additional actions are required that convert a physical size to a value of the events.

Note, that writing from SDF or CT model of computation might also trigger an event in the DE model of computation. However, this might not be useful or even cause problems. Usually, DE models are not activated by events at the data signals. DE models are usually activated by explicit control signals such as clock or enable signals, which is done by adapter classes that also provide such signals, and require an explicit controller.

For reading the queue of events that models the abstract signal, we use the following methods:

Reading from SDF ports: The value of the newest event is returned. For multi-rate dataflow, a conversion function is called which performs sample rate conversion. Sample rate conversion computes a weighted average of the values since the last call. Because time steps are constant, the time of the last call is the actual time minus the time step.

Reading from DE ports: The value of the newest event is returned.

Reading from CT ports: The value of the newest event is returned. This is applicable to the CT signal-flow model of computation, and for netlists. For netlists, additional actions are required that convert the abstract value to a physical size in a netlist.

Netlists and external simulators, Future work The implementation of polymorphic signals covers signals in SystemC-AMS, but not yet nodes in netlists. Actual work is to implement interfaces that support the coupling of external simulators, such as VHDL-AMS or SPICE, that also support simulation of netlists. For coupling external simulators of netlists, we extend the polymorphic signal using the following concept:

Netlists can write to polymorphic signals. This can be done by a port that leaves the netlist, and that specifies the physical sizes (e.g. a current, or a voltage), and its conversion to an abstract, non-conservative size. After conversion, this value is treated like a value from the CT signal-flow model of computation.

Netlists can 'read' from polymorphic signals, but require a small conversion circuit that is added to the netlist. This small conversion circuit is e.g. a current or voltage source, and is also specified by attributes at the port. Current work is to automatically insert such a circuit by a polymorphic signal to an external simulator.

4 Case studies

For evaluation we coose the design shown by Figure 7. It shows a control loop that controls the voltage of a power driver. The voltage is an average of pulses generated by a pulse generator. The voltage is measured, and a difference between the actual value and the programmed value is computed. Then, a PI controller computes a new pulse width. The left part of Figure 7 shows the executable specification, for which we used the continuous time model of computation. For the refinement of computation we exchanged single modules with modules that use other models of computation (CT-signal flow → SDF → DE). The right part of Figure 7 shows the design after refinement of interfaces.

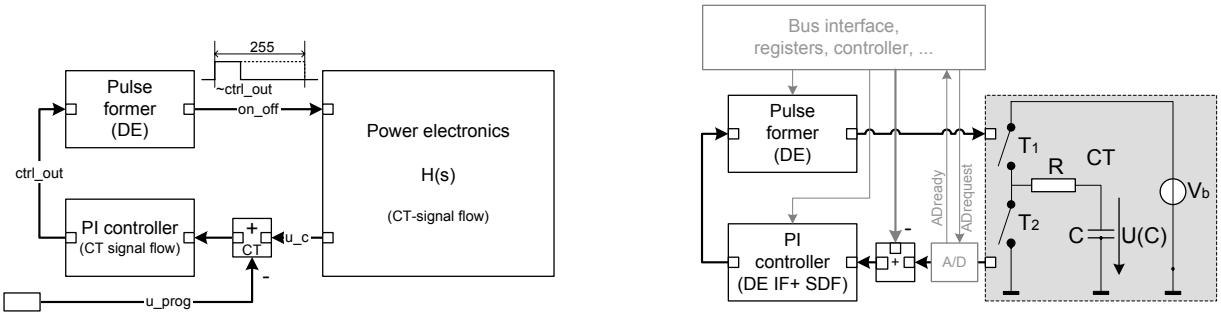

Figure 7: PWM driver: Executable specification (left), and after refinement (right).

The simulation of netlists that are coupled with polymorphic signals as shown in the right part of Figure 7 is not yet supported by polymorphic signals. However, this is the focus of current work.

Polymorphic signals were especially useful for the determination of sample rates and bit widths, which have an impact to the dynamic behavior. Figure 8 shows two different output signals that are produced by changing the sample frequencies and models of computation.

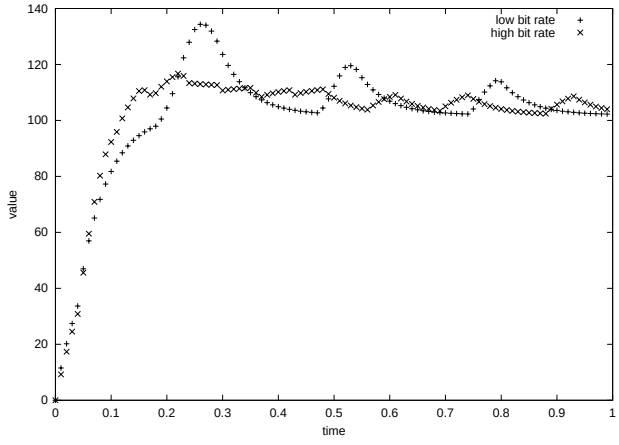

Figure 8: Output of PWM driver with different sample frequencies.

5 Discussion

SystemC provides a very general approach for modeling digital systems. It is a generic framework for different models of computation by changing the semantics of signals that are accessed via interfaces. However, all models of computation are simulated by a discrete event simulator. Models of computation that have different interfaces or value types can only be converted, if they are subtypes. SystemC-AMS extends SystemC for modeling analog and mixed-signal systems. In Ptolemy II, different models of computation are implemented by (different) directors, and behavioral types allow the coupling of different models of computation by considering the behavior of communication and synchronization. However, blindfold use without knowing the semantics might cause problems. Semantic types and polymorphic signals as proposed in this paper permit an implicit conversion of different types and automatically treat semantic issues in the right way, e.g. the conversion of value ranges or sample rate reduction in signal processing applications.

However, it is still unclear, if polymorphism is helpful, or if dynamically changing behavior might be dangerous. In first experiences, polymorphic signals have proven very helpful, and allowed us to model even complex systems with very little effort. Another advantage is that polymorphic signals support mixed-level simulation: One can easily exchange a single module of the executable specification with its implementation without the need to modify the structure of the overall system, or to write converters.

For the design of complex and heterogeneous applications a combination of (careful) use of behavioral types and semantic types could be attractive. Behavioral types are useful, when single data objects are exchanged at different levels of abstraction. Semantic types and polymorphic signals as proposed in this paper are useful if application specific knowledge is required for the conversion between different models of computation.

References

[1] Edward Lee, Stephen NeuenDorffer, and Michael Wirthlin. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems, and Computers*, June 2003.

[2] Alain Vachoux, Christoph Grimm, and Karsten Einwich. Towards analog and mixed-signal soc design with systemc-ams. In *IEEE International Workshop on Electronic Design, Test and Applications (DELTA'04)*, Perth, Australia, 2004.

[3] Christoph Grimm and Klaus Waldschmidt. KIR – A graph-based model for description of mixed analog/digital systems. In *European Design Automation Conference*, Geneva, Switzerland, September 1996.

[4] Christoph Grimm and Klaus Waldschmidt. Repartitioning and technology-mapping of electronic hybrid systems. In *Design, Automation and Test in Europe '98 (DATE)*, Paris, France, February 1998.

[5] Stuart Swan. An Introduction to System-Level Modeling in SystemC 2.0. Technical report, Open SystemC Initiative, 2001.

[6] Edward Lee and Yuhong Xiong. A Behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 2004.

[7] Christoph Grimm. Modeling and Refinement of Mixed Signal Systems with SystemC. In *SystemC – Methodologies and Applications*. Kluwer Academic Publisher (KAP), June 2003.

[8] Karsten Einwich, Peter Schwarz, Christoph Grimm, and Klaus Waldschmidt. Mixed-Signal Extension for SystemC. In Eugenio Villar and Jean Mermet, editors, *System Specification and Design Languages*. Kluwer Academic Publishers, Apr 2003.

[9] Christoph Grimm, Wilhelm Heupke, and Klaus Waldschmidt. Semi-Symbolic Modeling and Analysis of Noise in Heterogeneous Systems. In *Forum on Specification and Design Languages (FDL '04)*, Lille, France, September 2004.