

Interoperation of heterogeneous CAD tools in Ptolemy II

Jie Liu, Bicheng Wu, Xiaojun Liu, Edward A. Lee*

Department of Electrical Engineering and Computer Science

University of California, Berkeley, CA 94720, USA

ABSTRACT

Typical complex systems that involve microsensors and microactuators exhibit heterogeneity both at the implementation level and the problem level. For example, a system can be modeled using discrete events for digital circuits and SPICE-like analog descriptions for sensors. This heterogeneity exists not only in different implementation domains, but also at different levels of abstraction. This naturally leads to a heterogeneous approach to system design that uses domain-specific models of computation (MoC) at various levels of abstractions to define a system, and leverages multiple CAD tools to do simulation, verification and synthesis. As the size and scope of the system increase, the integration becomes too difficult and unmanageable if different tools are coordinated using simple scripts. In addition, for MEMS devices and mixed-signal circuits, it is essential to integrate tools with different MoC to simulate the whole system. Ptolemy II, a heterogeneous system-level design tool, supports the interaction among different MoCs. This paper discusses heterogeneous CAD tool interoperability in the Ptolemy II framework. The key is to understand the semantic interface and classify the tools by their MoC and their level of abstraction. Interfaces are designed for each domain so that the external tools can be easily wrapped. Then the interoperability of the tools becomes the interoperability of the semantics. Ptolemy II can act as the standard interface among different tools to achieve the overall design modeling. A micro-accelerometer with digital feedback is studied as an example.

Keywords: Heterogeneous simulation, models of computation, tool interaction, Ptolemy II

1. INTRODUCTION

Microelectromechanical systems (MEMS) are intrinsically complicated. A typical system consists of mechanical parts for sensing, analog circuits for pre-amplification, digital circuits for signal processing, microprocessor units and memories for configurable control functions, and embedded software. The increasing functional complexity suggests a top-down design methodology, where abstraction is a key concept. Higher level abstractions ease simulation and design space exploration.

This idea has been applied to digital circuit designs so successfully that a large number of digital systems are designed at an abstract behavioral level¹². Leveraging simulation, analysis and verification tools, libraries of previously built modules, and sophisticated synthesis algorithms, the design can migrate down levels of abstraction. The highest level of abstraction in a design flow is the system level, where the components of a system are represented in terms of their functionality or role in a design rather than in terms of their construction. As the design goes to lower levels, there are additional constraints to meet and less design space to explore. At the end of each stage, the design becomes more refined and closer to the physical implementation.

The current progress of MEMS technology, and vast design possibilities for large MEM systems, suggest top-down design³⁰. For example, in the first stage of designing an air-bag control system, once we can model the transfer function of a micro-accelerometer, we may not care whether it is a capacitive accelerometer or piezoelectric accelerometer, and we may not care about its geometrical dimension. What we may be more concerned about is the interface this accelerometer provides to the rest of the system. Is it analog or digital? What is the range of the signal? What sample rate is good for control?

Implementations of MEMS are heterogeneous. This suggests that designers to be able to model the components of the systems in different physical and computational domains. Heterogeneous design methodology¹⁰ aims to manage the complexity of the design process by using domain-specific models of computation hierarchically mixed and matched to define a system, and by seeking to find retargettable synthesis techniques from specifications to diverse implementation technologies. A *model of computation* (MoC) is the set of "laws of physics" that governs the interaction of the components in a model. Different models of computation may be suitable for different domains of modeling and different levels of abstraction. Components in the system may take distinct paths of refinement toward the physical design. Thus a challenge

* Correspondence: Email: {liuj, wbwu, liuxj, eal}@eecs.berkeley.edu; WWW: <http://ptolemy.eecs.berkeley.edu>

exists in how to verify at each design stage that the more refined component will still behave correctly in the whole system with other components.

Ptolemy II ²⁹ is a system-level design environment that supports heterogeneous modeling and design of concurrent systems. The design principle of Ptolemy II is that choosing the models of computation plays an important role in the process of designing complex embedded systems. The Ptolemy II software provides an infrastructure that allows designers to explore and integrate different models of computation to achieve the overall design goals. Different domains in Ptolemy II implement different models of computation. Since it is a system-level design tool, Ptolemy II does not provide the functionality for implementation-level simulation. But external tools based on different model of computation can be integrated into each domain, and Ptolemy II can serve as the semantic glue. This integration supports top-down design by providing a co-simulation environment in which the system can not only be simulated across models of computation, but also across levels of abstraction.

The rest of the paper is organized as follows. Section 2 discusses the heterogeneity of CAD tools. The Ptolemy II infrastructure and the interaction semantics of continuous-time models and discrete-event models are briefly discussed in section 3. The external tool interfaces in the two domains are given in section 4. Finally, a micro accelerometer with digital feedback is used as a case study to demonstrate how heterogeneous CAD tools are used in a top-down design flow.

2. HETEROGENEITY OF CAD TOOLS

As a result of decades of fast development of CAD technologies, a large number of tools have been developed to help the specification, simulation, and verification of various systems at various level of abstraction. Combining multiple tools in a design process is a natural way to design heterogeneous systems. There are usually two ways that multiple CAD tools are used in a design process.

- *Tools may be applied sequentially*, which means that the output of one tool is the input of another. In this case, the second tool is applicable if the result from the first tool provides enough information to execute the second tool. Typically, in a top-down design, after simulating the system at a higher level of abstraction, synthesis tools are employed to generate the specification of the system at a lower level of abstraction. This requires that the designer include enough synthesis information in the system specification.
- *Tools may be applied concurrently*, which means that multiple tools are integrated tightly in one stage of the design flow. For example, for a system that consists of both analog and digital parts, an analog simulator and a logic simulator may be employed at the same time to simulate the whole system. If there is feedback between the two parts, the correct interaction of tools is extremely critical.

The interoperability of tools in the first case has been studied in terms of design flow management (see e.g. [16]) or design frameworks (see e.g. [32]). This paper focuses on the interoperability of tools in the second case, especially when the tools implement different models of computation.

2.1. Overview

The MEMS community talks about energy domains like kinetic, elasticity, electrostatic field, magnetostatic fields, hydraulics, and heat, which directly reflect the physical implementation of the devices ¹⁷. For a system-level model, it may be inappropriate to include such physical details. It is the model of computation that characterizes a component in the system. Part of a MoC is the character of signals by which components communicate. For example, if a PDE formulation of the position of a beam under an electrostatic force is wrapped in a macromodel, then the signal at the boundary of the macromodel is a continuous-time signal. If the signal is sampled, then the interface to the rest of the system is discrete time.

There are a rich variety of models of computation that may be useful for modeling different parts of complex MEM systems. We outline some typical ones in this section.

- Continuous-time (CT) model. Ordinary differential equations (ODEs), differential algebraic equations (DAEs), and partial differential equations (PDEs) are in this category. The characterization of the model is that signals in the systems are continuous functions of a continuum that is interpreted as time. This is excellent for modeling analog circuits and many physical systems, and it is usually closely related to the implementation of the systems.
- Discrete-event (DE) model. The signals in this model are sets of events placed discretely in time. An event consists of a value and time stamp. This model of computation is popular for specifying digital hardware and simulating telecommunications systems, and has been realized in a large number of simulation environments, simulation languages, and hardware description languages, including VHDL and Verilog.

- Discrete-time (DT) model. Ordinary differential equations can be discretized to get difference equations, a commonly used model of computation in digital signal processing. A global clock defines the discrete points at which signals have values (at the ticks).
- Synchronous/reactive (SR) models³. Like the discrete-time model, signals are discrete in time. But unlike the discrete-time model, a signal need not have a value at every clock tick. SR models are excellent for applications with concurrent and complex control logic. Because of the tight synchronization, safety-critical real-time applications are a good match.

Other models of computation that can be found in designing embedded systems include finite state machines (FSMs), which are suitable for modeling sequential control logic, process networks¹⁵ (PN), which are suitable for modeling distributed systems and hardware architecture, and communicating sequential processes (CSP), which are suitable for modeling resources management in concurrent scenario. Dataflow is a special case of PN¹⁹.

Simulation tools usually work on one or more models of computation. The model of computation can be characterized by how they interpret time, how signal values vary with respect to time, and how the components in the system communicate. Below is a very incomplete list of tools categorized by their model of computation:

- Continuous-Time (PDE): Coyote¹¹, MARC²⁴;
- Continuous-Time (ODE/DAE): SPICE²⁶, Saber¹, Simulink²⁵;
- Discrete-Event: Digital circuit simulators, HDL behavioral simulators^{2,31};
- Dataflow: SPW⁶, COSSAP²⁸, HP-Ptolemy¹³;
- Synchronous/Reactive: Esterel⁵, Signal⁴, Lustre⁸, and Argos²³.

Discrete-event is the dominant model in digital circuit design. What MEMS devices and systems add to the traditional embedded system design is the heavy use of continuous-time models. This paper will focus on the semantics and tool interaction of these two models.

2.2 Continuous-Time and Discrete-Event models

- Partial Differential Equations (PDEs)

This is the model that is closest to the physical implementation. The signals of interest are functions of both time and space. Typical examples are stress models for mechanical structures, Maxwell's equations for electromagnetic fields, Navier-Stokes equations for fluids, and distributed-parameter models for thermal diffusion. Since they directly describe a physical system, they are tightly bound to an implementation, leaving few implementation options. Numerous PDE simulation tools have been developed for various physical models. The solving methods for PDEs are typically finite-element or boundary-element methods, which are quite costly. Most of the time only a small number of the critical parts in a MEM system need to be simulated under this model.

- Ordinary Differential Equations (ODEs)

Ordinary differential equations can be thought of as an abstraction of PDEs. That is why ODEs are called the macromodels for MEMS devices³⁰. ODEs have lumped parameters, and the signals in the system are only functions of time. ODEs can be specified using a block diagram syntax. The arcs represent continuous functions of a continuum that is interpreted as time. The blocks represent relations between these functions. The job of a simulator is to find a fixed-point, i.e., a set of functions that satisfy all the relations. The simulation of ODEs is still expensive compared to digital representations of comparable functionality.

There are two ways to specify a set of ODE, the conservative law model and the signal flow model¹⁴. The conservative law model defines a system by its physical components, which specifies relations of cross and through variables, and conservation laws are used to compile the component relations into a global system of equations. For example, in electrical circuits, the cross variables are voltages, the through variables are currents, and the conservation laws are the Kirchhoff's laws; in heat flows, the cross variables are temperature, the through variables are heat flow, and the conservative laws are the heat flow equations. This model directly reflects the physical components of a lumped-parameter system, thus is easy to construct from an implementation. The actual mathematical form of the system is hidden behind the scene. This is the model used in SPICE, Saber and most analog circuit simulators.

The signal-flow model is more abstract than the conservative law model. Entities in the system are maps that define the mathematical relation between input and output signals. Entities communicate by continuous-time signals. This model directly reflects the mathematical relations among signals, and is more convenient for specifying systems that do not have an explicit implementation yet. This is the model used in Simulink, for example.

The simulation of ODE models is performed by discretizing “the time continuum” into a discrete set of time points. The state and the outputs of the system are computed in chronological order. This is generally achieved by numerical integration methods, like linear multistep (LMS) methods or Runge-Kutta methods. Most simulation tools based on this model can adjust step sizes, control local errors, and handle stiff problems.

- **Discrete-Event Models**

Discrete-event (DE) models can also be given with a block-diagram syntax, where the arcs represent sets of events placed in time. An event consists of a value and time stamp¹⁸. The components in a system respond to input events and produce output events (simultaneously or in the future) that may trigger other parts of the system.

A typical discrete-event simulator operates by maintaining an event queue, in which the events are sorted by time stamp. During the simulation, the output events from all actors will be fed into the queue. At each iteration of the execution, the events with the smallest time stamp are removed from the queue, and the actors that respond the events are fired. The time stamp of the event that is just removed from the queue is defined as the current time of the system. An *iteration* of the simulation is defined as the processing all the simultaneous events at the current time. For the correctness of simulation, the actors in a DE system must be causal¹⁸, which roughly means the output events must not be earlier than the input events that trigger them.

3. MIXED-SIGNAL SIMULATION IN PTOLEMY II

3.1. Ptolemy II Infrastructure

Ptolemy II supports heterogeneous modeling and design of concurrent systems at the problem level. In Ptolemy II, a system is represented as a collection of hierarchical and concurrent components. The kernel of Ptolemy II defines a small set of Java classes that implement a data structure that supports uninterpreted clustered graphs. A graph consists of entities that have ports, plus relations that connect the entities through the ports. A hierarchical graph can be constructed by making an entity at one level of the graph contain another graph. And the hierarchies can be arbitrarily nested. Each cluster of the graph can implement different semantics, by realizing a model of computation that guides the execution of the graph.

While the kernel package only provides an abstract syntax, the actor package attaches the semantics of message passing and execution to it. This semantics is shared by a number of models of computation, including the continuous-time and the discrete-event models. The executable component in the actor package is called an Actor. An actor that does not contain any actors is an instance of the class `AtomicActor`, distinguishing it from a `CompositeActor`, which contains other actors. Messages are encapsulated in tokens. Actors have instances of `IOPort` which can be input ports, output ports, or both, depending on whether they can receive tokens, send tokens or both.

The `Executable` interface defines how an object can be invoked. An execution is defined to be one invocation of `initialize()`, followed by any number of iterations, followed by one invocation of `wrapup()`. The `initialize()` method is assumed to be invoked exactly once during the lifetime of an execution of an application. It may be invoked again to restart an execution. An iteration is defined to be one invocation of `prefire()`, any number of invocations of `fire()`, and one invocation of `postfire()`. The `wrapup()` method will be invoked exactly once per execution, when the execution terminates. The methods `initialize()`, `prefire()`, `fire()`, `postfire()`, and `wrapup()` are called the *action methods*.

Atomic actors implement the executable interface and add their own functionality. The execution of the components of a composite actor is governed by a director. Directors also implement the executable interface, and all the executable functions of a composite actor are delegated to its director. The director of the container of a composite actor is called the executive director for the composite actor. If a composite actor has no directors, then it is called transparent, and it relies on its executive director to execute. In general, the top-level composite actor (which has no containers) should have a director and no executive director.

Directors may implement different model of computation. If the director and the executive director of a composite actor are different, then two models of computation are integrated. In this situation, the interface between the two models must be carefully studied to insure a correct and efficient interaction.

3.2. Mixed-signal Simulation

By mixed-signal simulation, we refer to simulating a system that consists continuous-time parts (based on ODE modeling) and discrete-event parts in the same model. This is not trivial because the signals in the two models are intrinsically different. Extra work must be done to convert one type of signal to another. In addition, time is modeled differently in the two models. The execution control of the two subsystems must be carefully designed to make sure time is consistent in the two models.

Here we briefly introduce the interaction semantics designed in the Ptolemy II environment. For a detailed discussion, please refer to [22].

3.2.1. Continuous-time part

In Ptolemy II, we choose ordinary differential equations as the underlying mathematical model for continuous-time systems because it is widely used, has mature numerical solving methods, and the system has a deterministic (unique) solution under simple conditions. In addition, we choose the signal-flow model to be the interaction semantics among continuous-time actors. Conservative law semantics may be used within an actor to define its I/O relation. The signal flow model is more abstract than the conservative law model so that at the system level of the design, the designer does not need to worry about the implementation details of the components. On the other hand, conservative law models and event PDEs can be easily wrapped into a continuous-time actor, since the interface of those models are still continuous-time signals, which is compatible with the continuous-time semantics.

The existence and uniqueness theorem of ODEs (see e.g. [21]) allows the functions in a continuous-time system to be discontinuous at a countable number of discrete points, which are also called *breakpoints*. These breakpoints may be caused by a discontinuity in input signals or by the intrinsic property of some actors in the system. In theory, the solutions at these discontinuous points are not well defined. But the left and right limits at these points are. So instead of solving the ODE at those points, we would actually try to find the left and right limits.

A breakpoint may be known beforehand, in which case it is called an *expected breakpoint*. For example, a square wave source actor could tell its next flip time. This information can be used to control the discretization of time. A breakpoint can also be *unexpected*, which means that it is unknown until the time it occurs. For example, an actor that varies its functionality when the input signal crosses a threshold can only report a “missed” breakpoint after an integration step is finished. The following breakpoint handling mechanism can guarantee that no breakpoint is missed in the simulation.

Expected breakpoints are stored in a breakpoint table in chronological order. Before one integration step starting from t , if the intended step size is h (probably resolved from error control concerns), the director will query the breakpoint table for the earliest breakpoint, say b . If $t < b < (t + h)$, then the current step size is adjusted to be $b - t$. If $b = t$, which means that this is the first step after the breakpoint, then the breakpoint at t will be removed from the table. At the same time, the director may replace the current ODE solver with a breakpoint solver, and it may adjust the step size to the minimum step size.

After each integration step, say from t to $t + h$, the director will ask all actors that may generate unexpected breakpoint for “missed” breakpoints. If any one of those actors reports a “miss”, then the actor will be asked for a refinement of the step size, and the last integration step will be restarted with the refined step size. This process will be continued until the “missed” breakpoint is accurately located. Since the step sizes of integration steps are usually small, and the numerical results of the ODE are usually acceptable within a given error, the iterative mechanism for locating the unexpected breakpoints are generally not expensive.

3.2.2. Discrete-event part

A slight subtlety arises in discrete-event simulations with sources of events. Since the source actors have no input, there is no event in the event queue that can trigger its firing. This problem is solved in Ptolemy II (as well as in Ptolemy 0.x) using pure events⁹. A pure event is an event that has no value. Whenever the source actor is fired, besides emitting signal events, it puts a pure event into the event queue with itself as the destination and its next firing time as the time stamp. When this pure event is at the top of the event queue, the source actor can be refired. Of course, this requires that the actor be fired at least once at the beginning of the simulation. This refiring mechanism is also a key for mixed domain simulations.

3.2.3. Signal conversion

Event generators are actors that convert continuous signals to discrete events. The event at time τ with value v (we write $e(\tau, t)$) is defined by a trigger condition and a value evaluation. The trigger condition can be written as:

$$q(x, u, t) = 0 \tag{1}$$

where x is the state of the continuous-time system, u is the system input, and t is time. Once the event time is detected, say at τ , the value of the event is computed by

$$v = r(x(\tau), u(\tau), \tau) \tag{2}$$

If the q function in (1) depends only on time, then the event generator can know its next event time beforehand. It can just register its next event time, say t_k as a breakpoint with the director. When the current time reaches t_k , it can compute the

event value and register the next event time again. If the q function in (1) depends on the state of the system, then the event generator can only report a “missed” event after an integration step is finished. This is an unexpected breakpoint for the execution. Thus, the breakpoint mechanism in Ptolemy II CT domain can correctly handle the event generation problem.

Event interpreters are actors that convert discrete events into a continuous signal. The critical task for event interpreters is to provide a default value at the time points where no events occur. Event interpreters can be application specific. Usually, users may interpret events in the form of zero-order-hold signals or Dirac impulses. The zero-order-hold event interpreters are widely used because they are consistent with D/A converters. Not all continuous-time simulators are able to handle Dirac impulses correctly. In Ptolemy II, we implement the method presented in [33], which uses the backward Euler integration method twice (once forward in time and once backward in time) to resolve the correct state after the impulse.

3.2.4. Execution coordination

Multiple domains are integrated in Ptolemy II by the container-containee relationship. CT and DE are both timed domains, so when they are combined together, they must share the same notion of global time. The global time is maintained by the outer domain. The current time of a domain is accessible from the `getCurrentTime()` method of the director. A composite actor with an inside domain may have its own local time. But this local time must be correctly tuned such that the signals at the boundaries follows the semantics of the outer domain.

It has been shown that when a CT subsystem is contained in a DE domain, the local time of the CT subsystem should be ahead of the global time in the DE system²². This ahead-of-time execution implies that the CT subsystem should be able to remember its states and be able to rollback to an earlier time if the input event time is smaller than the current local time. The state it needs to remember is the state of the system after it has processed an input event. Consequently, the CT subsystem should not emit detected events before the global current time reaches the event time. Instead, it should request a refire from the executive director at the event time.

It is much simpler when the DE domain is embedded in the CT domain. The mechanism for breakpoint handling in the CT domain allows the DE subsystem to register its next output event as a breakpoint. Since time is advanced monotonically in CT, and the event generators can only generate events that are at the “current time,” the DE subsystem will receive inputs events monotonically in time. In addition, the causality of DE actors will ensure that the time stamp of the output events is always greater than or equal to the CT current time. That is, the DE subsystem only produces expected breakpoints.

4. TOOL INTERFACE IMPLEMENTATION

It can be seen from the above discussion that the interaction of CT and DE models is not trivial. Not all the CAD tools are designed for interacting with other tools, especially when the other tools implement other models of computation. For example, not all continuous-time simulation tools handle breakpoints, particularly unexpected breakpoints. Even fewer of them support the execution coordination with discrete-event simulators. This modeling mismatch makes the integration of different CAD tools very difficult.

On the other hand, it is usually easier for tools to interact with tools that support the same model of computation. Our approach for heterogeneous design crossing abstraction levels is to use Ptolemy II for semantic interface between two models of computation, and wrap external simulation tools only in the domain that has the same model of computation.

Generally, a simulation tool that can be wrapped as an actor should as least support the following operations.

- 1) `start()`. Start the tool as a separate process with preset options.
- 2) `loadDesign(design_file)`. Load the design to be simulated.
- 3) `setParameter(parameter_name, parameter_value)`. Assign the given value to the parameter with the given name in the external tool or the loaded design.
- 4) `setInput(input_name, input_value)`. Assign the given value to the input with the given name in the loaded design.
- 5) Setting a stop condition, for example, the simulation stop time.
- 6) `run()`. Start the simulation. The simulation will stop once the stop condition is met.
- 7) `getOutput(out_name): output_value`. Extract the output value from the tool.
- 8) `exit()`. Terminate the tool process.

Many operations, like 3) to 7) may be domain dependent. Other requirements may also be needed for specific domains. These operations may be supported by the tools in two forms, a command line interface or a programming language interface (PLI). In a command line interface, a set of command is provides to access the internal information of the simulation tool. In a

programming language interface, these functions are provided by shared libraries. Certain programming languages, e.g. C, may use these libraries to build new applications.

Ptolemy II employs external simulation tools by wrapping them into atomic actors. Such an actor has input ports, output ports, and parameters. The input token and parameters are converted to the parameter or internal variables of the tool simulation. The results from the tool simulation are converted to output tokens and sent out from the output ports. The action methods of the actors implement the semantics of a certain domain based on the operations listed above. In general, the action methods are implemented as:

- initialize(): starts the tool, and reads design files.
- prefire(), fire(), postfire(): assigns input data and parameter to the tool, runs the simulation, and processes the result from the simulation. The detailed tasks performed within the prefire(), fire(), and postfire() methods depend on the model of computation of the domain where the tool is interfacing with.
- wrapup(): terminates the tool.

For tools that provide command line interface, the action methods are implemented as follows. In the initialize phase of the execution, the external tool is started as a separate process, and its standard input and output pipes are obtained by the actor as I/O streams, as shown in Figure 1. Thus controlling the tool execution is simplified to writing to the input stream, and reading from the output stream.

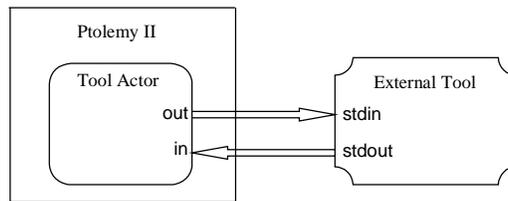


Figure 1. Command line interaction with external tools.

For a programming language interface (PLI), a wrapper can be written in the same language as required by the PLI. The wrapper provides a command line interpreter for the library. For each interaction command, the wrapper will translate it to a set of PLI methods call. Thus the tool interface actor communicates, by commands, with the wrapper, as shown in Figure 2.

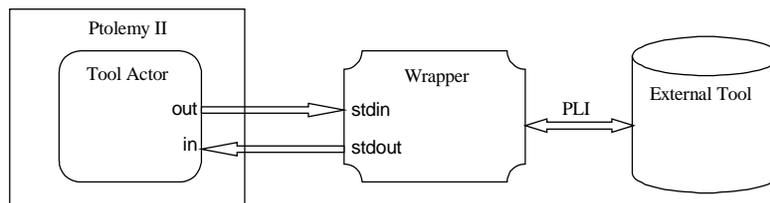


Figure 2. PLI interaction with external tools.

4.2. CT Domain Tool Actor

CT tool actor should behave as an atomic actor in a CT system. For each execution step in a CT system, the tool actor performs a transient analysis on the loaded design for that time period. In particular, a CT tool actor does not perform major operations in the prefire() or postfire() methods. The pseudo code for the fire() method is shown in Figure 3.

```

fire(){
    read input data from input ports ;
    assign input data to corresponding variables in the subsystem;
    set the end time of the firing;
    simulate the subsystem from the previous saved state to the end time;
    process the output stream and produce output token(s);
    save the current state;
}

```

Figure 3. Pseudo code for the fire() method for the tool actor in the CT domain.

In addition, for correct interacting with event-based simulations, an external simulation tool in the CT domain should be able to save its state (all the independent variables in the simulation, including time) and restore the saved state when needed. Then the rollback operation of the CT domain can be performed by saving and restoring states.

4.3. DE Domain Tool Actor

The DE tool actor should behave like a DE composite actor. The external tool should obey the global time maintained by the DE system. At each iteration of the DE domain, the external tool should process all the events at the “current time.” Ideally, the tool should provide the next event time in its local event queue. For example, a function `cliGetNextEventTime()` is provided in VSS VHDL Simulator²⁸. After a tool actor is fired, it registers its next event time as a pure event to the DE director one level up. For tools that does not support the query of the next event time, techniques also exist to integrate discrete-event simulators with or without rollback²⁷.

The `prefire()` and `postfire()` methods of a DE tool actor do no major operation. The pseudo code for the `fire()` method is shown in Figure 4.

```

fire() {
    read input data from input ports, if there is any;
    assign input value, if they exists, to netlist variables and registers;
    set the end time to be the current time of the director;
    start the simulation;
    process the output result, and decide whether there are output events;
    output the token(s) if necessary;
    query the tool for the next event time, and request refiring at that time.
}

```

Figure 4. Pseudo code for the `fire()` method for a tool actor in the DE domain.

A DE tool actor does not always produce the an output token, for example, it might decide only to generate an output when there is a rising edge on the clock, or falling edge on the clock.

5. CASE STUDY

We explore the top down design of a micro-accelerometer with sigma-delta digital feedback²⁰. Beams and anchors, separating by gaps, form parallel plate capacitors. When the device is accelerated in the sensing direction, the displacement of the beams causes the change in the gap sizes, which causes the change of the capacitance. By measuring the change in capacitance (using the Winston capacitor bridge), the acceleration can be obtained accurately. Feedback can be applied to the beams by charging the capacitors. Using feedback can reduce the sensitivity to process variations, eliminate mechanical resonances, and increase sensor bandwidth, selectivity and dynamic range.

Sigma-delta modulation⁶, also called the pulse density modulation or bang-bang control, is a digital feedback technique. It gets the A/D conversion functionality “for free”. Figure 5 shows the conceptual diagram of system. The central part of the digital feedback is a one-bit quantizer.

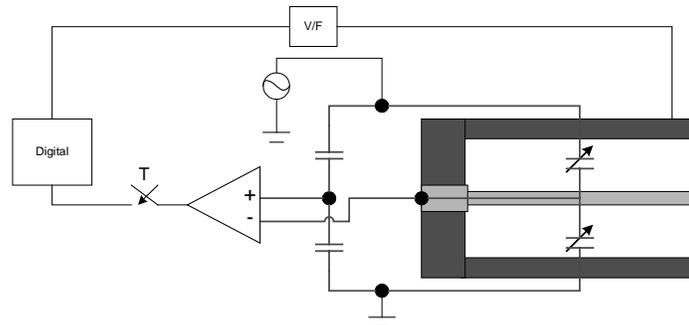


Figure 5. Micro-accelerometer.

At the system level, the system is modeled using a continuous-time model for the beam and discrete-event model for the digital circuit. This is the CT/DE part in Figure 6. The parameters adjusted include sampling time, FIR filter tags, and feedback gain.

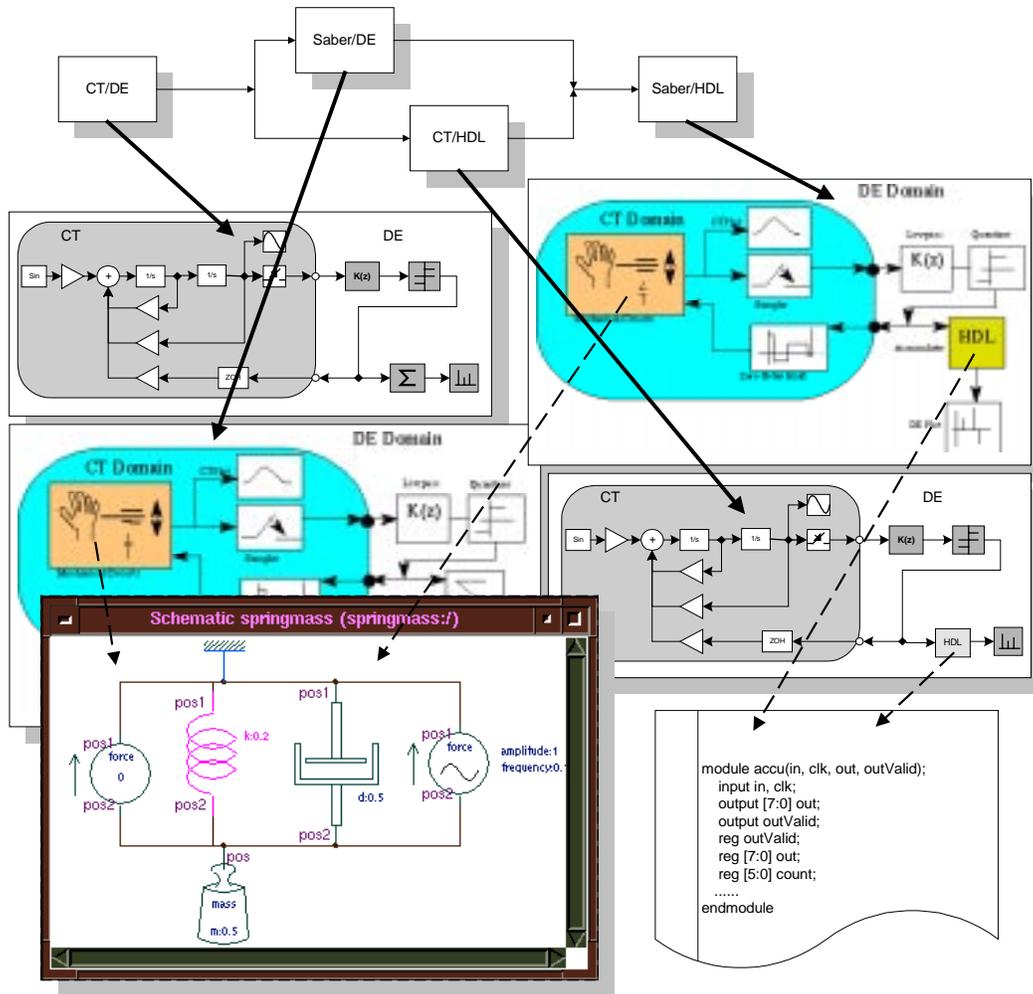


Figure 6. System-level block diagram for the micro-accelerometer.

After the system level simulation, the design moves one level down, and the analog and digital parts are separately modeled in Saber and Verilog. The Saber model is shown at the bottom-left of Figure 6, where the beam is modeled as a spring-mass system with damping. At this level, we can add noise to the system and study the sensitivity and robustness of the control. The accumulator is modeled using Verilog, since this part is better implemented as a synchronous digital circuit. The design parameters are clock rate and noise reduction in the accumulator (implemented as a low-pass filter). The system is simulated using Saber/Ptolemy, and Ptolemy/DE. Finally, the whole system is simulated by integrating Ptolemy II, Saber and Verilog.

6. CONCLUSION

This paper discusses the interaction of heterogeneous simulation CAD tools in the Ptolemy II environment. By studying the model of computation that a CAD tool implements, we can have a deeper insight into how tools should interact. Integrating multiple CAD tools then becomes the study of the interaction among models of computation. The paper discusses the interaction of continuous-time models and discrete-event models. The result shows that a correct interaction is not trivial, and not all domain-specific CAD tools support this interaction. Ptolemy II, which implements the interaction semantics among different MoC, can serve as the glue for heterogeneous tools. The integration of Saber and Verilog is studied as an example.

7. ACKNOWLEDGMENTS

The authors would like to thank Analogy Inc. for providing the Saber™ software. This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, Motorola, NEC, and Philips.

8. REFERENCE

1. Analogy Inc., <http://www.analogy.com>
2. P. J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers, 1996.
3. A. Benveniste and G. Berry, "The synchronous Approach to Reactive and Real-Time Systems," *Proc. of the IEEE*, Vol. 79, No. 9, pp. 1270-1282, 1991.
4. A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Tran. on Automatic Control*, Vol. 35, No. 5, pp. 525-546, May 1990.
5. G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, Vol. 19, No. 2, pp. 87-152, 1992.
6. Cadence Design Systems, Inc. <http://www.cadence.com>
7. James C. Candy, "A Use of Limit Cycle Oscillations to Obtain Robust Analog-to-Digital Converters," *IEEE Trans. on Communications*, Vol. COM-22, No. 3, pp.298-305, March 1974.
8. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A Declarative Language for Programming Synchronous Systems," *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, Munich, Germany, January, 1987.
9. W.-T. Chang, S. Ha, and E. A. Lee, "Heterogeneous Simulation-- Mixing Discrete-Event Models with Dataflow," *Journal on VLSI Signal Processing*, Vol. 13, No. 1, Jan. 1997.
10. W.-T. Chang, A. Kalavade, and E. A. Lee, "Effective Heterogeneous Design and Cosimulation," chapter in *Hardware/Software Co-design*, G. DeMicheli and M. Sami, eds., NATO ASI Series Vol. 310, Kluwer Academic Publishers, 1996.
11. Coyote Systems, Inc., <http://www.coyotesystems.com>
12. S. Devadas, A. Ghosh, and K. Keutzer, *Logic Synthesis*, McGraw-Hill, 1994.
13. Hewlett-Packard Com., <http://www.hp.com>
14. IEEE DASC 1076.1 Working Group, "VHDL-A Design Objective Document, version 2.3", http://www.vhdl.org/analog/ftp_files/requirements/DOD_v2.3.txt
15. G. Kahn and D. MacQueen, "Coroutines and Networks of Parallel Processes," *Info. Proc.* 77, Vol. 7, pp. 993-998, 1977.
16. A. Kalavade, "System Level Codesign of Mixed Hardware-Software Systems," *UCB/ERL Memorandum 95/88*, Ph.D. Dissertation, Dept. of EECS, University of California, Berkeley, CA 94720, Sept., 1995.
17. G. Kovacs, *Micromachined Transducers Sourcebook*, WCB McGraw-Hill, 1998.
18. E. A. Lee, "Modeling Concurrent Real-time Processes Using Discrete Events," *UCB/ERL Memorandum M98/7*, Dept. of EECS, University of California, Berkeley, CA 94720, March 4th 1998.
19. E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proc. of the IEEE*, Vol. 83, pp. 772-801, May 1995.
20. M. A. Lemkin, "Micro Accelerometer Design with Digital Feedback Control," Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Fall 1997.
21. D. Lemon, *Differential Equations*, Prentice-Hall, 1988.
22. J. Liu, "Continuous Time and Mixed-Signal Simulation in Ptolemy II," *UCB/ERL Memorandum M98/74*, Dept. of EECS, University of California, Berkeley, CA 94720, Dec. 15th 1998.
23. F. Maraninchi, "The Argos Language: Graphical Representation of Automata and Description of Reactive Systems," *Proc. of the IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.
24. MARC Analysis Research Corp., <http://www.marc.com>
25. The MathWorks, Inc., <http://www.mathworks.com>
26. D. Pederson, "A Historical Review of Circuit Simulation," *IEEE Trans. on Circuits and Systems*, Vol. CAS-31 No.1, pp.103-111, Jan 1984.
27. W. Sung and S. Ha, "Optimized Timed Hardware Software Cosimulation without Roll-back," *Design, Automation and Test in Europe (DATE'98) Conference Proceedings*, Paris, France, pp.945-946, Feb 23-26, 1998.
28. Synopsys, Inc., <http://www.synopsys.com>
29. The Ptolemy Group, "Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java," <http://ptolemy.eecs.berkeley.edu/ptolemyII>
30. S. Senturia, "CAD Challenges for Microsensors, Microactuators, and Microsystems," *Proc. of the IEEE*, Vol. 86, No. 8, pp.1611-1626, Aug. 1998.
31. D. E. Thomas, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 1998.
32. P. van der Wolf, *CAD Frameworks: Principles and Architecture*, Kluwer Academic Publishers, 1994
33. J.Vlach and A. Opal, "Modern CAD Methods for Analysis of Switched Networks," *IEEE Trans. on Circuits and Systems -I: Fundamental Theory and Applications*, Vol. 44, No. 8, Aug. 1997.