Journal of Circuits, Systems, and Computers

ACTOR-ORIENTED DESIGN OF EMBEDDED HARDWARE AND SOFTWARE SYSTEMS

EDWARD A. LEE*
EECS Department, University of California at Berkeley,
Berkeley, California 94720, U.S.A.

STEPHEN NEUENDORFFER*
EECS Department, University of California at Berkeley,
Berkeley, California 94720, U.S.A.

MICHAEL J. WIRTHLIN†
ECEN Department, Brigham Young University,
Provo, Utah 84602, U.S.A.

Invited paper, *Journal of Circuits, Systems, and Computers*
Version 1, July 8, 2002

In this paper, we argue that model-based design and platform-based design are two views of the same thing. A platform is a set of designs. Model-based design is about using platforms with useful modeling properties to specify designs, and then synthesizing implementations from these specifications. A design language, such as SystemC or VHDL, defines a platform. Any valid expression in the language is an element of the set. Viewed from below, the language provides an abstraction of implementation capabilities. Viewed from above, the language provides a set of constraints together with benefits that flow from those constraints. It provides a conceptual framework within which a design is crafted. An actor-oriented platform lies above the program-level platforms that are widely used today. It orthogonalizes the actor definition language and the actor assembly language, enabling domain-polymorphic actor definitions, multiple models of computation, and hierarchical heterogeneity. Actor-oriented platforms offer compelling modeling properties. Synthesis into program-level descriptions is possible. We illustrate these concepts by describing a design framework built on Ptolemy II.

Keywords: actor-oriented design, embedded systems, model-based design, synthesis, Ptolemy II, SystemC, JHDL

1. Introduction

Embedded systems interact with the physical world through sensors and actuators. These days, most include both hardware and software designs that are specialized to the application. Conceptually, the distinction between hardware and software, from the perspective of computation, has only to do with the degree of concurrency and the role of time.

---

[*] Lee and Neuendorffer are supported in part by the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of California MICRO program, and the following companies: Agilent Technologies, Cadence Design Systems, Hitachi, National Semiconductor, and Philips.

[†] Wirthlin is supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Los Alamos National Laboratory.

An application with a large amount of concurrency and a heavy temporal content might as well be thought of using hardware abstractions, regardless of how it is implemented. An application that is sequential and has no temporal behavior might as well be thought of using software abstractions, regardless of how it is implemented. The key problem becomes one of identifying the appropriate abstractions for representing the design.

Unfortunately, for embedded systems, it is rare that a single unified approach to building such abstractions will work. HDLs with discrete-event semantics are not well-suited to describing software. On the other hand, imperative languages with sequential semantics are not well-suited to describing hardware. Neither is particularly good at expressing the concurrency and timing in embedded software.

The design community has been systematically addressing this problem by increasing the expressiveness of the languages in use. VHDL, for example, offers a reasonably expressive imperative subset that can, in principle, be used to design software. To attempt to unify the design styles, the VLSI design community has made heroic efforts to translate imperative VHDL into hardware (using so-called *behavioral compilers*), with only limited success. Moreover, VHDL lacks key features important to software designers, such as object orientation. More importantly, the concurrent side of VHDL has discrete-event semantics, which is effective for RTL descriptions of hardware, where concurrency translates into parallel hardware, but not so effective for software, where concurrency translates into multitasking. So VHDL remains a hardware description language.

A significantly different direction has been to develop domain-specific languages and synthesis tools for those languages. For example, Simulink, from The MathWorks, was originally created for control system modeling and design, and has recently come into significant use in embedded software development (using Real-Time Workshop, and related products), and experimentally in hardware design [9]. Simulink supports an instance of what we call *actor-oriented design*. We will define this precisely in section , but loosely, actors are concurrent components that communicate through ports and interact according to a model of computation.

VHDL offers designers an instance of what we call *program-level design*, which will be discussed below. Interestingly, it also offers an actor-oriented abstraction, where processes are actors that communicate via discrete-event signals. This abstraction plays a major role in the synthesizable subset. SystemC, which originated as Scenic[31], similarly offers an actor-oriented abstraction with the context of program-level design. Its processes define actors communicating with (mostly) synchronous semantics. These processes can be synthesized into synchronous circuits if the synthesizable subset is used[40]. JHDL[19] offers a similar approach, but within the context of the Java language rather than the C++ language.

All three program-level design approaches (synthesizable VHDL within VHDL, SystemC within C++, and JHDL within Java) closely bind the actor-oriented abstraction with the program-level abstraction. This is distinct from Simulink, for example, where the language used to define blocks is orthogonal to the block interaction semantics (the model of computation). In Simulink, blocks can be defined in M (the Matlab scripting language), C, Ada, or Fortran, at least. We will argue that separating the actor-oriented abstraction from the program-level abstraction brings some key benefits. This separation will likely

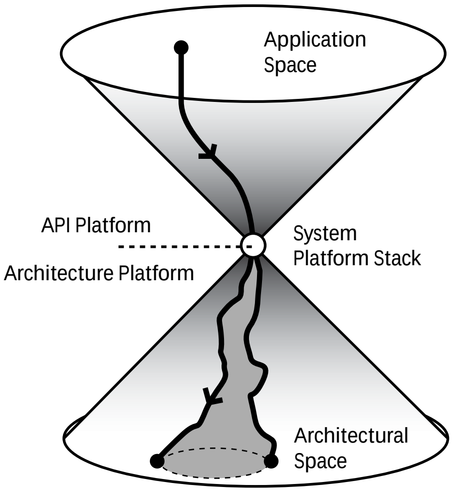

Fig. 1. Alberto's Hourglass.

dominate future design practice.

In this paper, we attempt to step back from the specific tools and languages and examine the role of actor-oriented design and program-level design in embedded hardware and software systems.

2. Platform-based and Model-based design

It has been frequently recognized that building designs more abstractly (with fewer implementation details) makes them more understandable, and therefore allows designers to build more complex systems. Two approaches that have gained a great deal of attention recently are "platform-based design," popularized by Alberto Sangiovanni-Vincentelli and colleagues [22,38], and "model-based design," popularized by Janos Sztipanovits and colleagues[41]. We will argue that these are two views of the same thing, and will give a conceptual framework for understanding their basic tenets.

2.1. *Platform-based design*

Figure 1 is a representation that Sangiovanni-Vincentelli frequently uses to explain platform-based design. At the top is the "application space," which is a set of designs. An application instance is a point in that space. The downward arrow from this space represents a mapping by the designer of an application into an abstract representation that conforms with the constraints of the platform. The lower arrows represent (typically automatic) mappings of that abstract representation into concrete designs in the platform. The upper half is called the "API platform" and the lower half the "architecture platform." The bottleneck (vertices of the cones) represents the constraints of the platform meeting the conceptual model within which the designer builds the design.

Inspired by this, give a somewhat more formal structure here. We define *platform* to be a set. We will call elements of the set *designs*. Examples of such sets are:

- The set of all x86 binaries.
- The set of syntactically correct Java programs.
- The set of all Java byte-code programs.
- The set of standard-cell ASIC designs using a given cell library.
- The set of all synthesizable VHDL programs.
- The set of all digital CMOS integrated circuits.
- The set of all Wintel PCs.
- The set of all Wintel PC applications.
- The set of all ANSI C programs.
- The set of all FPGA configurations for a Xilinx Virtex II XC2V4000

The value in a platform is the benefits that arise from working with a restricted set of designs. For example, synthesizable VHDL programs are synthesizable, unlike general VHDL programs. ANSI C programs can be compiled to run on just about any computer.

For a platform to be useful, a designer must be able to recognize when a design is a member of a platform. Many less successful efforts use, for example, a “subset of C” to define silicon circuits, but fail to define precisely the subset that works. A subset is a new platform.

For each platform, there are two key (but separable) issues:

1. How the set is defined.

2. How the elements of the set are described.

For example, the set of all Java byte-code programs is defined by the Java virtual machine specification. A member of this set is defined by a finite sequence of Java byte codes, typically stored in a class file or a collection of class files. An ill-defined “subset of C” makes it clear how elements of the set are described, but not how the set is defined. Given a C program, one cannot tell whether it is a member of the set.

The hourglass in figure 1 does not clearly separate these two issues, so we prefer a representation like that in figure 2. In this representation, we replace the bottleneck with a platform. The bottleneck is a reference to how elements of the upper platform are described. We instead think of the middle layer as consisting of the set of all possible designs in some application specification language. In figure 2, the upper region is a set of applications (a platform), for example the set of all audio signal processing systems. The middle region is the set of designs (also a platform) in some specification language, for example the set of all C programs. The lower region is a set of “architectural” designs (also a platform), for example the set of all x86 binaries. The arrows represent the same things as in figure 1, but now as relations between sets. The two lower arrows represent, for example, that two distinct compilers may produce distinct binaries from the same C program. The shaded area in the architectural space represents, for example, the set of all valid x86 binaries that execute the give C program.

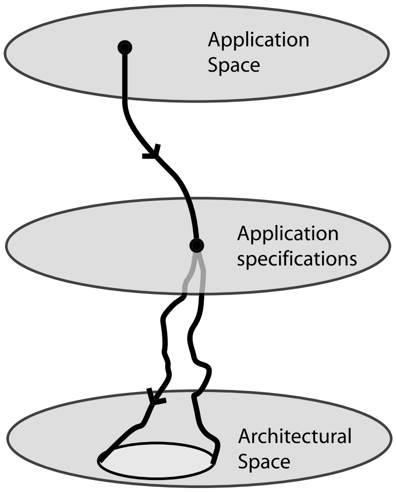

Fig. 2. Three platforms and mappings between them.

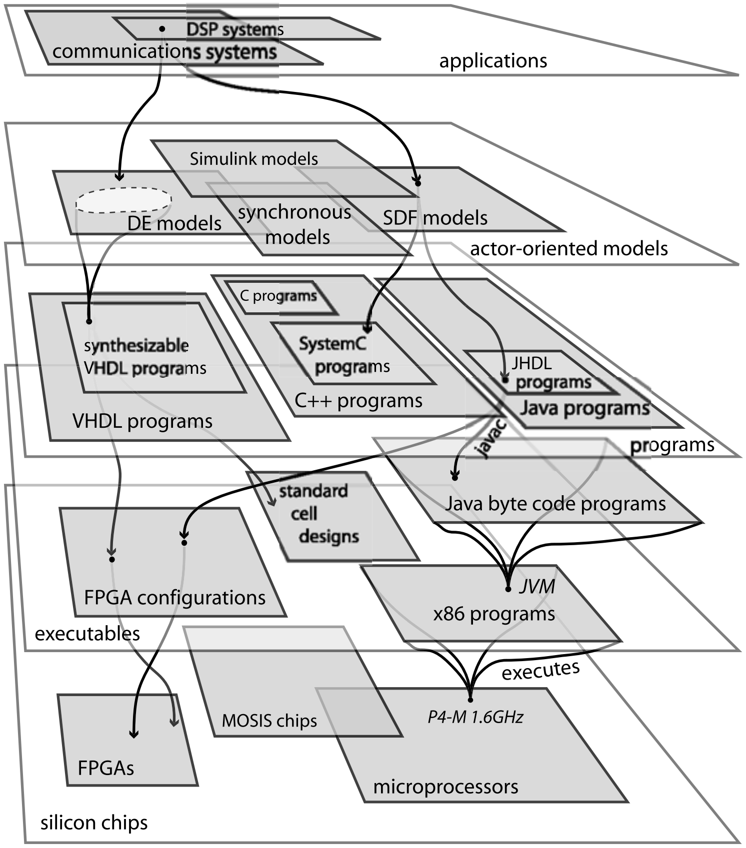

Fig. 3. Platforms and mappings between them.

A *relation* R from platform P_1 to P_2 is a subset of $P_1 \times P_2$. P_1 is called the *domain* and P_2 the *codomain* of R. Relations between platforms play a key role in design. A *function* $F\colon P_1 \to P_2$ is a relation $F \subset P_1 \times P_2$ where

$$(p_1, p_2) \in F \text{ and } (p_1, p_3) \in F \Rightarrow p_2 = p_3.$$

Functions that map one platform into another are realized by, for example, synthesis tools and compilers.

The key difference between figures 1 and 2 is that in figure 2, we do not attempt to describe how the members of the sets (the dots in the platforms) are represented. Of course, the efficacy of a platform will depend there being a reasonable and manipulable representation, and on there existing one or more relations with platforms that are closer to a physical system. Figure 2 now has a formal structure, that of sets and relations. This structure can be manipulated independent of semantics. More importantly, it makes it clear that for a given application, there may be many application models. For example, we can subdivide the application specification layer into more refined, domain-specific platforms.

We can now give a much more complete picture that lends insight into the roles of tools and platforms. In figure 3, each box is a platform. It is a set of designs. The arrows between boxes represent mappings (functions), which convert one design into another. For example, the Java compiler *javac* converts a member of the set *Java programs* into a member of the set *Java byte-code programs*. The platforms are stacked roughly by degree of abstraction from a physical realization (we will give a more precise meaning to this stacking shortly). There are two key concerns:

1. It may be possible to map a single design in one (higher) platform into several designs in a (lower) platform.
2. It may be possible for several designs in a (higher) platform to map into the same design in a (lower) platform.

(We put “higher” and “lower” in parentheses pending provision of a formal reason for putting a platform higher or lower.) For example, the dot labeled “P4-M 1.6GHz” (an Intel processor) induces a subset of *executables* labeled *x86 programs*, which is a subset of executables that it can execute. Consider a function *executes*,

$$\textit{executes}\colon \textit{x86 programs} \to \textit{microprocessors}$$

where, as illustrated in figure 3,

$$\forall\, x \in \textit{x86 programs}, \quad \textit{executes}(x) = \textit{P4-M 1.6GHz}.$$

This function represents the fact that the P4-M 1.6GHz processor can execute any x86 binary. In fact, the P4-M 1.6GHz itself becomes a useful platform (a set with only one member), where the function *executes* induces another platform *x86 programs*. This connection between a physically realizable platform (the P4-M 1.6GHz processor) and an abstract one (the set of x86 programs) is essential to being able to use the abstract platform for design. Moreover, this connection has to not just exist as a mathematical object (the function *executes*), but it has to be realizable itself (and of course, it is). Thus, platforms that are useful for design must have paths via realizable relations to physically realizable platforms.

This begins to address the question: if a platform is merely a set of designs, how do we distinguish a good platform from a bad one? Sangiovanni-Vincentelli defines platform as follows[38]:

> "... an abstraction layer in the design flow that facilitates a number of possible refinements into a subsequent abstraction layer (platform) in the design flow."

In our structure, this is more a description of a platform that is *useful* (for design) than a definition of the concept of *platform*. To be more precise, we need to combine platforms and relations:

> A *design framework* is a collection of platforms and realizable relations between platforms where at least one of the platforms is a set of physically realizable designs, and for any design in any platform, the transitive closure of the relations from that design includes at least one physically realizable design.

A relation $R \subset P_1 \times P_2$ is *realizable* if for all $p_1 \in P_1$, there is a terminating procedure (manual or automatic) that yields $p_2 \in P_2$ such that $(p_1, p_2) \in R$. "Physically realizable design," however, is a term we leave undefined, at least formally. In figure 3, any design in the lowest platform *silicon chips* is physically realizable.

"Transitive closure" can be formally defined. Given a collection of platforms $P_1, \cdots, P_N$ and relations $R_1, \cdots, R_M$ between platforms, we say that two designs p_1 and p_2 are *transitively related* if there exist elements $r_1, \cdots, r_Q$ of $R_1, \cdots, R_M$ such that $r_1 = (p_1, a_1)$, $r_2 = (a_1, a_2)$, $\cdots$, and $r_Q = (a_{Q-1}, p_2)$. The transitive closure from a design p_1 is the set of all designs that are transitively related to p_1.

Figure 3 illustrates (incompletely) a design framework where communication systems or DSP applications are manually translated into models obeying either discrete-event (DE), synchronous dataflow (SDF), or Simulink semantics (all of which are actor-oriented). This manual translation process is represented formally as a relation R. Consider a member of this relation,

$$(x, y) \in R \subset \textit{applications} \times \textit{actor-oriented models}.$$

We interpret this member to mean that model y realizes application x (making this any more precise would require formalizing what we mean by "realizes," which would be challenging). Whether this relation is "realizable" (by a designer) depends on many factors, some very fuzzy, such as how good the user interface is to design tools like Simulink or SDF block diagram editors. Indeed, focusing on realizable relations between these two top layers is a valid and challenging research area.

In figure 3, synthesis tools (a set of relations that happen to be functions) are used to generate Java programs, C programs, or VHDL programs from the actor-oriented models.

These are then compiled or synthesized into FPGA configurations, standard-cell designs, binary programs, or Java byte code. The set *Java byte code programs* bears a relation with one or more specific designs in *x86 programs* (for example), which realize a byte code interpreter. The JVM interpreter (and any other x86 program) then bears a relation to members of the set *microprocessors*. This completes the path to a physically realizable platform.

The platform *Java byte code programs* lies between the platforms that we call *executables* and *programs*. This in-between layer is newly popular, and represents the virtual machine concept. Indeed, the trend has been towards adding layers to the platform stack. The actor-oriented layer is relatively new, still immature, largely unproven, and the focus of this paper.

For any two platforms, we can define a relation by pairs of designs that we consider in some sense to be equivalent. They realize the same system, at least with respect to some aspect of the behavior of that system that might be of interest. For two platforms P_1 and P_2, a *refinement relation* $R \subset P_1 \times P_2$ is a relation where $\forall\, p_1 \in P_1$, $\exists\, p_2 \in P_2$ such that $(p_1, p_2) \in R$. That is, to be a refinement relation from P_1 to P_2, then for every design in P_1 there must be at least one "equivalent" design in P_2.

Consider a relation $R \subset P_1 \times P_2$. We write the *image* of a point $p_1 \in P_1$ as $I_R(p_1)$, and define it to be the largest subset of P_2 such that

$$p_2 \in I_R(p_1) \Rightarrow (p_1, p_2) \in R.$$

For a refinement relation, the image of a design p_1 is the set of all possible refinements. The shaded squiggly paths at the bottom of figure 1 is intended to represent members of a refinement relation. The shadow they cast in the lower platform is the image of the design.

For the same relation $R \subset P_1 \times P_2$, we define the *coimage* $C_R(p_2)$ of a point $p_2 \in P_2$ as the largest subset of P_1 such that

$$p_1 \in C_R(p_2) \Rightarrow (p_1, p_2) \in R.$$

For a refinement relation, the coimage of a point is the set of all designs that can be refined into it. In figure 3,

$$C_{\textit{executes}}(\textit{P4-M 1.6GHz}) = \textit{x86 programs}.$$

Refinement relations might have strong properties that bind two platforms in useful ways. We define for example a *strict refinement* to be a refinement relation $R \subset P_1 \times P_2$ that satisfies the following constraint:

$$\text{if } (p_1, p_2) \in R \text{ and } (p_1', p_2) \in R, \text{ then } I_R(p_1) = I_R(p_1').$$

This says that if two designs in P_1 are equivalent to the same design in P_2, then they have exactly the same set of refinements.

For example, consider a refinement relation T between the set of *Java programs* and the set *Java byte code programs* based on the Turing test. That is, if p_1 is a Java program and p_2

is a byte code program, then $(p_1, p_2) \in T$ if the input/output behavior of the two programs is identical. For any particular Java program, there are many byte code programs that are Turing equivalent to that program. Different Java compilers, or even the same compiler with different command-line options, will generate different byte code realizations of the program.

Such a refinement relation is strict if the following statement is true: If Java programs p_1 and p_1' compile into the same byte code p_2 (using any compiler, or even distinct compilers), then every byte code that is a legitimate compilation of p_1 is also a legitimate compilation of p_1'.

That a refinement relation is strict is useful. It means that if the set *Java byte code programs* has a semantics, then that semantics, together with the strict refinement relation, induces a semantics on the set *Java programs*. That is, if byte code has a meaning, then so does Java code. A framework where every design is transitively related to a physically realizable design via strict refinement relations is a particularly useful framework. It means that every design in the framework has a meaning with respect to the physically realizable designs.

The semantics induced by a strict refinement relation may be incomplete. A trivial case, for example, is the refinement relation labeled "executes" in figure 3. This relation is a function that maps every x86 program into a single point in *silicon chips*, the P4-M 1.6GHz. The semantics induced on x86 programs, therefore, is fairly weak. All it says that any x86 program executes on a P4-M 1.6GHz. However, a strict refinement relation between Java programs and byte code programs (a compiler) is not so weak. Indeed, this relation can be used to define an operational semantics for Java in terms of the operational semantics for byte code.

In figure 1, it is implied that for every design in the "application space" there are many designs in the "architecture space." This represents the notion that the design in the application space is more "abstract" than the one in the architecture space. This notion of abstraction does not appear to be formal, but rather to reflect the concept that there exist mappings of designs in one platform into designs in another that can result in several designs in the latter.

Figure 3 stacks the platforms vertically, also trying to capture the notion that designs above are more "abstract" than those below. A "top-down" design process would begin at the top and use refinement relations (that are preferably strict and realizable by some computer program) to refine the design to one that is physically realizable. Determining how abstract a design is then becomes a simple matter of determining how many refinement relations separate the design from one that is physically realizable. By this metric, platforms in figure 3 that are higher are more abstract.

2.2. *Model-Based Design*

A *model* is a design bearing a particular relationship to another design, the one being modeled. Suppose $p_1 \in P_1$ is a model for $p_2 \in P_2$. To be a useful model, we require that if some statement about p_1 is true, the some closely related statement about p_2 is also true.

For example, we could construct a synthesizable VHDL program p_2, and then construct a performance model $p_1 \in$ *DE models*. That performance model might discard functionality, and talk only about timing. Furthermore, if p_1 satisfies certain timing properties, then we infer that so does p_2.

To be useful for modeling, a platform has to have useful modeling properties. In particular, it needs to be possible (and preferably easy) to determine whether a property of interest is true for a particular design in the platform.

Model-based design[41] is simply the observation that if one uses a modeling language to state all the important properties of a design, then that model can and should be refined into an implementation.

To accomplish model-based design, therefore, one needs a design framework. To call it model-based design, one needs for the specification constructed by the designer to be in a language that expresses what is important about the design. The "modeling language" defines a platform (the set of all designs expressible in the language). Moreover, to be useful, the platform has to be effectively realizable. That is, it must function as a platform in platform-based design.

For platform-based design to be useful, designs in a platform must express what is important about the system being designed, and preferably no more. Thus, model-based design and platform-based design are essentially two sides of the same coin. The term "platform-based design" is typically used when looking at platforms from below, emphasizing the refinement relations to physically realizable platforms. The term "model-based design" is typically used when looking at a platform from above, emphasizing refinement relations between the application space and the platform.

From the perspective of a semiconductor vendor, therefore, platform-based design is the way to go. This vendor needs to provide customers with platforms and refinement relations into their semiconductor technology. The design of these platforms needs to reflect the capabilities and limitations of the technology. From the perspective of a communications systems engineer, however, model-based design is the way to go, since it is based on platforms that reflect the constraints and requirements of the application space and enable effective engineering analysis. Of course, for a communications engineer to attempt a system implementation, the perspectives of both platform-based design and model-based design are crucial. Platforms are needed that reflect the capabilities and limitations of less abstract platforms into which they will be refined, and that offer convenient and understandable sets of designs to those that have to create the designs.

A key issue (and one that is largely unresolved) is that in modeling, multiple views are often useful. That is, one might construct a performance model to examine timing, and a rather different model to examine power consumption in some design of an electronic system. This is suggested by the light area in the set *DE models* in figure 3, which is the coimage of a design in the set *synthesizable VHDL programs*, and reflects that fact that there is often more than one useful DE model of a particular chip. Although there has been some success with multi-view modeling[27,23], it still seems unclear how to generally and systematically blend multiple abstract models to create a refinement.

We address a more modest problem, and one that we believe is largely resolved: heterogeneous design. We will focus on the set of actor-oriented models in figure 3, and show that by carefully defining the platforms within this set, we can achieve the objectives of both model-based design (designer convenience) and platform-based design (synthesizability). The key mechanism becomes a hierarchical form of heterogeneity and an abstract semantics that makes this hierarchical heterogeneity possible. Before we tackle this, however, we will address current efforts to improve design practice at the program level.

3. Program-Level Design

In figure 3, the *programs* platform shows three subplatforms, VHDL, Java, and C++. Each of these contain a subplatform that has been created for hardware design, namely synthesizable VHDL, JHDL, and SystemC. SystemC has a further subset for RTL design[40]. We call this strategy of defining a subset of a generic programming language for synthesis *program-level design*. Interestingly, all three of these subplatforms offer actor-oriented abstractions, but bind these abstractions with the program level. We will argue that separating the actor-oriented abstraction from the program-level abstraction offers compelling advantages. We will discuss SystemC in detail because it represents the major current effort in the hardware design community.

3.1. *SystemC*

SystemC (see http://systemc.org) is a recent attempt from the hardware community to define a new abstraction the bridge the gap between software and hardware design methodologies. The effort began with the Scenic design environment[31], which became SystemC 1.0[30]. Scenic adopted a mainstream software language, C++, and defined a class library that offered concurrent semantics suitable to hardware design. Thus, in figure 3, SystemC is shown as a subset of C++ (all SystemC programs are C++ programs, but not vice-versa).

In a first effort to define concurrent semantics more suitable to software than that of VHDL, Scenic emphasized cycle-driven models rather than discrete-event models. On the hardware side, this has the effect of limiting the designs to synchronous circuits. On the software side, it makes execution of the concurrent system dramatically more efficient than what is possible with discrete-event simulation, opening the possibility of using these concurrent designs directly as deployable software.

In Scenic, each component has its own thread of execution. Upon construction of a component, a user specifies a clock signal to which the component is sensitive. Upon changes in value of this clock signal, the thread associated with the component is awakened for incremental processing. That thread executes until it next calls wait(), which stalls the thread until the next tick of the clock. Scenic also provides a wait_until() method, which stalls the thread until a specified condition on a signal is satisfied, and a watching() method, which causes a call to wait() or wait_until() to return with an exception when a specified condition on a signal occurs. Scenic also allows for multiple asynchronous clocks, and schedules the ticks of the clocks using a simplified discrete-event scheduler. Taken together, these innovations offer a much stronger alignment with software practice (through the use of C++) and much better execution efficiency (through the use of cycle-driven models).

The first of these brings hardware designs closer to software, and the second of these makes concurrent software designs look more like hardware. The second, in fact, offers an actor-oriented abstraction.

A significant issue in Scenic is that since each component conceptually has its own thread of control, naive implementation can result in significant context switching overhead when realizing designs in software. Scenic cleverly reduces this overhead for components that do not process data on every clock tick through the use of sensitivity and watch lists, which avoid awakening processes unless there is real work to be done. Thus, it leverages the formal properties of the (synchronous and cycle-driven) model of computation to gain simulation efficiency. In addition, SystemC 1.0 extends Scenic by adding *method*-style components, where a component is triggered not by awakening a thread, but instead by invoking a method. Conceptually, the component process becomes a sequence of invocations of this method, and maintenance of the state of the component becomes the responsibility of the component, rather than the responsibility of the thread program counter. Again, the ability to do this depends critically on the model of computation, which is an essential part of the actor-oriented abstraction.

However, the synchronous signal communication in Scenic and SystemC 1.0 has been viewed as too limiting for system designers. SystemC 2.0[39] addresses this concern by augmenting the SystemC model with *channels*. A channel is an object that serves as a container for communication and synchronization. Channels implement one or more interfaces (access methods) that can reflect specialized properties of the communication style that is used for the particular channel. For example, Swan[39] describes a fixed-length FIFO queue channel with blocking reads and writes that can be reset by the producer, but not by the consumer. This approach is similar to that in Metropolis[6], where concurrent components communicate via protocols that can be chosen by the designer. This approach has been called "interface-based design"[37]. The intent in SystemC 2.0 is that by using a set of channels from a library, a designer can build a concurrent design using communication constructs that are abstract from a particular hardware realization.

We fear that this approach fails to solve its principal motivating problems, as given by Liao[31], "the lack of a single language in which to design hardware and software." Although nominally there is a single language syntax, C++, the interaction between components is conceptually very different from C++. It is actor oriented. Furthermore, although SystemC provides a abstraction of the communication between components that enables some simple communication refinement, this interaction is relatively unstructured, limiting the innate modeling properties of the designs. The designer has great flexibility in structuring component interaction, because that component interaction is specified at a low level, by explicitly describing its mechanism. This flexibility has a price, however. It makes designs harder to understand and analyze, and it increases execution costs.

We address first the execution costs. To allow for a reasonably rich set of concurrent interactions between components, SystemC 2.0 emphasizes that each component has its own thread of control. Correct execution must implement mutual exclusion on all access methods for all objects. In Java, this would be realized by requiring all methods of all objects to be declared `synchronized`. In C++, it requires use of a thread package to implements such mutual exclusion. The Java experience indicates, however, that mutual exclusion of this type is quite expensive, adding considerably to the already substantial context switching costs of a multi-threaded style of design. This mutual exclusion is required only because the model of computation imposes few constraints on the structuring of component interaction.

We address next the observation that flexible component interactions, given at a low level, make designs harder to understand and analyze. Through its channels, SystemC 2.0 will tempt designers to mix and match communication styles. For instance, they might combine buffered, asynchronous message passing with mailboxes and rendezvous. We call the result an *amorphous amalgam*. It will be difficult to understand, and resulting designs will be difficult to validate.

SystemC 3.0, which does not exist yet, appears to be moving further in the direction of components being associated with their own thread of control[7]. The objective is a literal representation of RTOS processes as components, and an augmentation of the component model to reflect the properties of an RTOS process. Operating system features, such as execution priorities and thread creation and destruction, become part of the component model. The result, it seems likely, will be a C++ class library that supports a range of system-level design, from RTL descriptions of synchronous hardware components through to processes executing on a CPU under the control of an RTOS. All of these processes must be properly synchronized during execution for correct language implementation.

Furthermore, the features of RTOS processes are very different from the features of hardware processes, as are the ways they are used in the design process. A system designer will likely be required to decide early in the design process what executes in hardware and what executes in software. Hardware portions of a design will use lightweight components from a synthesizable library interconnected by synchronous signals and using methods, and software portions will use heavyweight processes with carefully designed message-passing channels to handle the communication between them.

Ultimately, it seems that component interaction is where language support is really needed, rather than in low level expressions and iterative statements. Component interaction is far more difficult to understand and to get right in a design than those aspects of the design that have been homogenized through the use of C++. Properly structured component interaction can also greatly improve execution performance by enabling sequentialized software execution without operating system processes, as was done in the original Scenic design environment. Unfortunately, the useful properties of specialized models of computation have been lost in the "generic model of computation"[39] of SystemC.

Specifically, while SystemC 2.0 offers a framework for building designs using abstract event-driven communication, it stops short of realizing any useful model-based abstraction. SystemC 3.0, with its (current) focus on RTOS processes, also does not seem likely to address this problem. A designer that chooses the semantics of each communication channel individually faces a severe challenge in getting the design right. The interactions between the various communication semantics will be very difficult to understand. Classical pitfalls, such as priority inversion, where synchronization locks due to communication may interfere with scheduling policy, will look trivially easy by comparison to the mysterious deadlocks, livelocks, and timing anomalies that are likely to result.

To fix this (rather severe) problem, considerable discipline will be required on the part of the designer. In particular designers will need to follow the precepts of a specific model of computation to get understandable, predictable, and robust designs, but must do so without any help from the language. If, for example, determinate computation is important in a design consisting of asynchronous communicating threads, then it would be wise to follow the constraints of Kahn and MacQueen[21] and avoid using any component whose behavior depends on whether there is input available on a particular channel at a particular time. This requires considerable discipline in a language where it is so easy to define a channel with an interface including a `num_available()` method[39], which returns the number of available data items on a channel. Similarly, the reset() method available in the same channel example will have a highly nondeterministic effect in an asynchronously executing framework. It will take a rather sophisticated designer to recognize these pitfalls without leveraging models of computation. Even when using predefined libraries of appropriately abstract channels, we fear that it is simply too easy to break the constraints of a model of computation when such constraints must be ensured explicitly by a designer. Most designs will end being an amorphous amalgam.

Instead of the “generic model of computation”[39] of SystemC 2.0, we propose supplying designers with a generic, actor-oriented platform to specify designs, as in figure 3. Synthesis tools can be used to map these designs into more detailed, architecture-specific SystemC descriptions. These patterns of transformation are not easily manipulated by designers, but can be performed automatically by design tools. We have prototyped a system that does exactly this. For experimental convenience, this system produces a Java description of software and a JHDL description of hardware.

4. Actor-Oriented Design

Actor-oriented modeling is a component methodology that is particularly effective for system-level design (close to the application level in figure 3). Components called *actors* execute and communicate with other actors in a *model*, as illustrated in figure 4. Actors have a well defined component interface. This interface abstracts the internal state and behavior of an actor, and restricts how an actor interacts with its environment. The interface includes *ports* that represent points of communication for an actor, and *parameters* which are used to configure the operation of an actor. Often, parameter values are part of the *a priori* configuration of an actor and do not change when a model is executed. The configuration of a model also contains explicit communication *channels* that pass data from one port to another. The use of channels to mediate communication implies that actors interact only with the channels that they are connected to and not directly with other actors.

Like actors, which have a well-defined external interface, models may also define an external interface, which we call its *hierarchical abstraction*. This interface consists of *external ports* and *external parameters*, which are distinct from the ports and parameters of the individual actors in the model. The external ports of a model can be connected by channels to other external ports of the model or to the ports of actors that compose

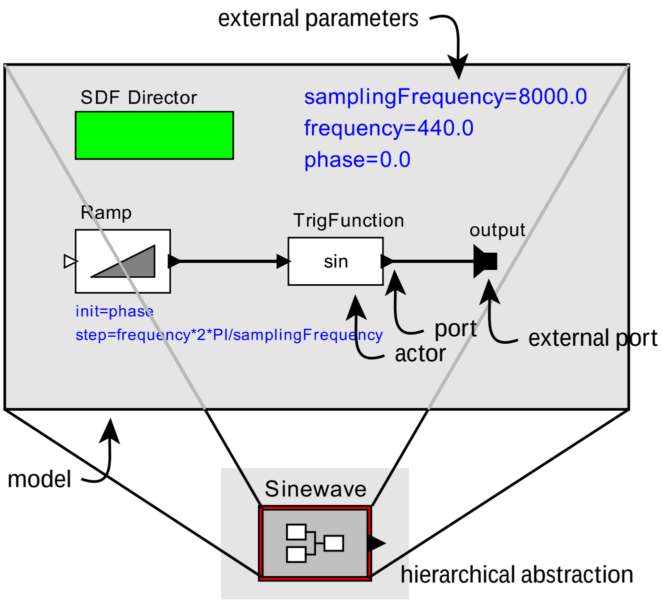

Fig. 4. Illustration of an actor-oriented model (above) and its hierarchical abstraction (below).

the model. External parameters of a model can be used to determine the values of the parameters of actors inside the model.

Taken together, the concepts of models, actors, ports, parameters and channels describe the *abstract syntax* of actor-oriented design. This syntax can be represented concretely in several ways, such as graphically, as in figure 4, in XML[29] as in figure 5, or in a program designed to a specific API (as in SystemC). Ptolemy II[8] offers all three alternatives.

It is important to realize that the syntactic structure of an actor-oriented design says little about the semantics. The semantics is largely orthogonal to the syntax, and is determined by a *model of computation*. The model of computation might give operational rules for executing a model. These rules determine when actors perform internal computation, update their internal state, and perform external communication. The model of computation also defines the nature of communication between components.

The utility of a model of computation is the possibility of formally described *derived properties* that apply to all similar models. Depending on the model of computation, the model may be provably determinate[20], statically schedulable [28], or time safe[17]. Because of its derived properties, a model of computation represents a style of modeling that is useful in any circumstance where those properties are desirable. In other words, models of computation form *design patterns of component interaction*, in the same sense that Gamma, *et al.* describe design patterns in object oriented languages[13]. It is exactly these derived properties that make actor-oriented models a useful approach to model-based design, since they correspond to desired properties of the model. We summarize a few useful models of computation for hardware and software design.

```
<class name="Sinewave">
    <property name="samplingFrequency" value="8000.0"/>
    <property name="frequency" value="440.0"/>
    <property name="phase" value="0.0"/>
    <property name="SDF Director"
        class="ptolemy.domains.sdf.kernel.SDFDirector"/>
    <port name="output"><property name="output"/></port>
    <entity name="Ramp" class="ptolemy.actor.lib.Ramp">
        <property name="init" value="phase"/>
        <property name="step"
               value="frequency*2*PI/samplingFrequency"/>
    </entity>
    <entity name="TrigFunction"
            class="ptolemy.actor.lib.TrigFunction">
        <property name="function" value="sin"
            class="ptolemy.kernel.util.StringAttribute"/>
    </entity>
    <relation name="relation"/>
    <relation name="relation2"/>
    <link port="output" relation="relation2"/>
    <link port="Ramp.output" relation="relation"/>
    <link port="TrigFunction.input"
          relation="relation"/>
    <link port="TrigFunction.output"
          relation="relation2"/>
</class>
```

Fig. 5. An XML representation of the sinewave source.

4.1. *Synchronous/reactive models of computation*

The synchronous/reactive (SR) model of computation is actor-oriented, where the computation of actors is triggered by clocks, and at least conceptually, the computation is instantaneous and simultaneous in all the actors [2]. Most synchronous/reactive languages give a fixed-point semantics to resolve zero-delay cycles. Examples of such languages include Esterel[3], Lustre[14], and Signal[26]. These languages do not associate a separate thread of control with each component, and a compiler can compile away the concurrency, reducing a highly concurrent program to a single thread of control. Consequently, they are well suited to realization in software. Moreover, because of their hardware-inspired concurrency model, they have proven effective as specification languages for hardware. In practice, these languages are used at the actor-oriented level of figure 3, and are typically compiled into VHDL or C programs.

Compared to SystemC 2.0 or Metropolis, synchronous/reactive languages are highly specialized. There is only one type of communication, unbuffered and instantaneous with fixed-point semantics. This communication is integral to the semantics, and the strong formal properties of synchronous/reactive models flow from this semantics. For example, because of the semantics, it is possible (and often practical) to check models by exhaustively searching the reachable states of the model for undesirable states. Moreover, highly efficient execution is possible (for hardware simulation, for instance, using cycle-driven simulation[15], and for software design, by compilation that removes all concurrency [3]).

Time-triggered models of computation are closely related to synchronous/reactive ones. These models of computation have appeared as platforms at the lower levels of figure 3 (as hardware architectures) and at the higher levels (as actor-oriented languages). The time-triggered architecture(TTA)[25] is a hardware architecture supporting such models. The TTA takes advantage of this regularity by statically scheduling computations and communications among distributed components. The Giotto language [16] elevates this concept the actor-oriented level by defining a language that is compiled into more traditional programming languages for realization in real-time software.

Discrete-time models of computation are also closely related. These are commonly used for digital signal processing, where there is an elaborate theory that handles the composition of subsystems. This model of computation can be generalized to support multiple sample rates. In either case, a global clock defines the discrete points at which signals have values (at the ticks).

4.2. *Dataflow*

Despite the name, the synchronous dataflow (SDF) model of computation [28] is not synchronous in the same sense as synchronous/reactive models of computation. It is a dataflow model of computation. In dataflow models, actor computations are triggered by the availability of input data. Connections between actors represent the flow of data from a producer actor to a consumer actor, and are typically buffered. Examples of actor-oriented languages that use the synchronous dataflow model of computation are SPW (signal processing worksystem, from Cadence) and LabVIEW (from National Instruments).

SDF is also highly specialized. There is only one type of communication, and actors are required to obey a fixed contract that dictates the amount of data that they produce and consume when they execute. This specialization yields formal properties that are useful from both the modeling and synthesis perspectives. For example, SDF models can be statically scheduled. Moreover, their memory requirements can be determined statically, unlike more general dataflow models. And whether the model deadlocks can also be determined statically, unlike more general dataflow models. Like synchronous/reactive models, SDF has proven effectively realizable in both software and hardware. Design frameworks with SDF use it at the actor-oriented level in figure 3, and compile SDF specifications into VHDL, C, or some other language.

There are several richer dataflow models of computation. Boolean dataflow (BDF) is a generalization that sometimes yields to deadlock and boundedness analysis, although fundamentally these questions remain undecidable[5]. Dynamic dataflow (DDF) uses only run-time analysis, and thus makes no attempt to statically answer questions about deadlock and boundedness [36]. In Kahn process networks (PN) [20], actors execute asynchronously and communicate via FIFO queues. PN has been used effectively for actor-oriented design of signal processing systems[10].

4.3. *Discrete events*

In discrete-event (DE) models of computation, the connections between actors represent sets of events placed on a time line. An event consists of a value and time stamp (or just a time stamp, for *pure events*). This model of computation governs the process interaction through signals in VHDL and Verilog, and is used in Scenic and SystemC to link synchronous islands with asynchronous clocks. It is also used at the modeling level in a number of modeling packages aimed at analyzing telecommunication systems and queuing systems.

DE models are typically fairly literal descriptions of physical systems, and hence are somewhat less abstract than the other models of computation considered here. The notion of time in these models is very much the Newtonian physical notion of time, although with embellishments (such as delta delays in VHDL) to handle non-physical issues such as simultaneous events.

A main advantage of discrete-event models is that events can occur with almost any time stamp. In particular, it is simple to realize the notion of a component that has delay associated with it; the component simply creates output events at the appropriate point in the future. This is true even if the delay is unknown beforehand or is random. As such, discrete-event models have seen significant application in modeling digital logic circuits. Unfortunately, this advantage is often a disadvantage in some circumstances. Because delays are easy to change, it is often difficult to model the effect of simultaneous events. Additionally, modifications to one part of a design can easily affect other portions, since the ordering of events can easily be disturbed.

4.4. *Continuous time*

Physical systems can often be modeled using coupled differential equations. These have a natural representation as actor-oriented models, where the connections represent continuous-time signals (functions of the time continuum). The components represent relations between these signals. The job of an execution environment is to find a fixed-point, i.e., a set of functions of time that satisfy all the relations. Two distinct styles are used in practice. In one style, an actor defines a functional relationship between input signals and output signals. This style is realized for example in Simulink, from The MathWorks. In another style, an actor also defines a relation between signals, but no signal is considered an input or an output. The actor instead asserts constraints on the signals. This style is realized in Spice and many derivative circuit simulators.

Continuous-time (CT) models, like DE models, are typically fairly literal descriptions of physical systems, and hence are somewhat less abstract than the other models of computation considered here. The notion of time in these models is again the Newtonian physical notion, and again there are embellishments.

4.5. *Hierarchical Heterogeneity*

Of the levels shown in figure 3, actor-oriented design is the least mature. A large number of exploratory and commercial tools have been created and have evolved. Typically, these tools begin rather specialized, and become more general over time by enriching the semantics of their model of computation. This is not necessarily the best approach, however, because enriching the semantics can lead to loss of formal properties, thus hindering both the modeling objectives and the effective realizability of the designs. Less analysis and less optimization is possible. One may end up trying to analyze or realize designs that are extremely unstructured.

For a particular application, an appropriate model of computation does not impose unnecessary constraints, and at the same time is constrained enough to result in useful derived properties. Thus, for example, Scenic[31] specialized to synchronous designs, enabling cycle-driven simulation[15], which greatly improves execution efficiency over more general discrete-event models of computation (such as that found in VHDL). While a cycle-driven model of computation emphasizes periodic behavior, the synchronous dataflow model of computation [28] says nothing about timing, describing only functional data dependency. It leaves scheduling and timing details largely up to the synthesis tool, allowing it to co-optimize memory usage and execution latency[42]. Selecting an appropriate model of computation for a particular application is often difficult, but this is a problem we should embrace instead of avoiding.

Countering the desire for a specialized model of computation that is finely tuned to the needs of the application is the fact that applications are heterogeneous and complex. Models of multi-vehicle control systems[24], high-energy astrophysics experiments[34], and even simple control systems [12,33] can include continuous-time dynamics, multiple modes of execution, extensive signal processing, and distributed real-time execution. In such cases, it is difficult to model the heterogeneous aspects of a system effectively using a single, specialized model of computation. For instance, while the discrete-event model of computation used in VHDL is effective at representing discrete logic in an embedded system, it cannot capture the continuous-time aspects. While the problem can be avoided by requiring the designer to manually construct discrete approximations, it is generally more effective to use the continuous-time model of computation in cooperation with the discrete model of computation.

Many models of computation are semantically rich (e.g. Turing complete) and thus can express the operations of another model of computation. For example, this is the intent of the channels of SystemC 2.0. A better approach is to allow designers to use multiple, specialized models of computation, composed hierarchically.

Hierarchical heterogeneity enables the description of a heterogeneous application without selecting a single model of computation. It allows for the description of portions of a design using different models of computation. The basis for the composition is an abstract semantics that captures not the complete semantics of a model of computation, but only those aspects that are important for composition.

5. Abstract Semantics

We are trying to preserve the specialization of models of computation and also achieve generality. We are doing this by composing models hierarchically and heterogeneously. What makes this composition possible is an *abstract semantics*, which abstracts how communication and flow of control work. The abstract semantics is (loosely speaking) not the union of interesting semantics, but rather the intersection. It is abstract in the sense that it represents the common features of models of computation as opposed to their collection of features.

A familiar example of an abstract semantics is represented by the Simulink S-function interface. Although not formally described as such, it in fact functions as such. In fact, Simulink works with Stateflow to accomplish a limited form of hierarchical heterogeneity through this S-function interface. We will describe an abstract semantics that is similar to that of Simulink, but slightly simpler. It is the one realized in the Ptolemy II framework for actor-oriented design.

In Ptolemy II models[8], a *director* realizes the model of computation. A director is placed in a model by the model builder to indicate the model of computation for the model. For example, an SDF director is shown visually as the uppermost icon in figure 4. The director manages the execution of the model, defining the flow of control, and also defines the communication semantics.

When a director is placed in a model, as in figure 4, that model becomes an *opaque composite actor*. To the outside environment, it appears to be an atomic actor. But inside, it is a composite, executing under the semantics defined by the local director. Obviously, there has to be some coordination between the execution on the outside and the execution on the inside. That coordination is defined by the abstract semantics.

The flow of control and communication semantics are abstracted by the *Executable* and *Receiver* interfaces, respectively. These interfaces define a suite of methods, the semantics of which are the abstract semantics of Ptolemy II. A receiver is supplied for each channel in

initialize	Initialize the actor.
prefire	Test preconditions for firing and return true if firing can proceed.
fire	Read inputs and produce outputs.
postfire	Read inputs and update the state of the actor.
wrapup	End execution of the actor and free system resources.

Fig. 6. The key flow of control operations in the Ptolemy II abstract semantics.

a model by the director; this ensures that the communication semantics and flow of control work in concert to implement the model of computation.

5.1. *Abstract Flow of Control*

In the Ptolemy II abstract semantics, actors execute in three phases, *initialize*, a sequence of *iterations*, and *wrapup*. An iteration is a sequence of operations that read input data, produce output data, and update the state, but in a particular, structured way. The operations of an iteration consist of exactly one invocation of *prefire*, followed by zero or more invocations of *fire*, followed by zero or one invocation of *postfire*.

These operations and their significance are summarized in figure 6. The first part of an iteration is the invocation of *prefire*, which tests preconditions for firing. The actor thus determines whether its conditions for firing are satisfied. If it indicates that they are (by a return value of true), then the iteration proceeds with one or more executions of *fire* followed by exactly one invocation of *postfire*. These latter two operations can read (and possibly consume) input data values, but only *fire* can produce outputs.

If *prefire* indicates that preconditions are satisfied, then most actors guarantee that invocations of *fire* and *postfire* will complete in a finite amount of time. Such actors are said to realize a *precise reaction* [32]. A director that tests these preconditions prior to invoking the actor, and fires the actor only if the preconditions are satisfied, is said to realize a *responsible framework* [32]. Responsible frameworks coupled with precise reactions are key to hierarchical heterogeneity.

It is also expected of an actor that only *postfire* updates the state of the actor. That is, the *prefire* and *fire* operations are purely functional. This allows a director to iterate executions of *fire* of a family of actors in search of a fixed point. This can be used, for example, to solve algebraic loops (as done in Simulink), to iterate under the control of a numerical integration algorithm (also as done in Simulink), or to iterate to a fixed point in a cyclic synchronous/reactive model. In Ptolemy II, not all actors obey this contract (particularly hierarchically heterogeneous actors), and thus, not all actors can be placed within models that iterate to a fixed point. It is an ongoing research issue to design and realize actors that are assured of obeying this contract.

An example of an actor definition that provides these methods is shown in figure 7. This actor implements a counter that begins with a value given by its “init” parameter, and on each iteration, increments by the value given by the “step” parameter. Since it obeys the abstract semantics, it can be used in models using any model of computation that conforms to this abstract semantics. Such an actor is called a *domain polymorphic* actor in Ptolemy

```
public class Ramp extends TypedAtomicActor {

    public IOPort output = new IOPort(this, "output", false, true);
    public Parameter init = new Parameter(this, "init", new IntToken(0));
    public Parameter step = new Parameter(this, "step", new IntToken(1));

    public void initialize() {
       _stateToken = init.getToken();
    }

    public boolean prefire() {
       // Always ready to fire.
       return true;
    }

    public void fire() {
       // Send current state on channel 0.
       output.send(0, _stateToken);
    }

    public boolean postfire() {
       // Polymorphic add.
       _stateToken = _stateToken.add(step.getToken());
       // Indicate that firing can continue to the next iteration.
       return true;
    }

    private Token _stateToken;
}
```

Fig. 7. A specification for a simplified Ramp actor in Ptolemy II (simplified to ignore exception handling).

II terminology. The key to hierarchical heterogeneity is to ensure that composite model, like that in figure 4, and is itself a domain-polymorphic actor.

5.2. *Abstract Communication*

The abstract semantics provides the set of primitive communication operations shown in figure 8. These operations allow an actor to query the state of communication channels, and subsequently retrieve information from the channels or send information to the channels.

These operations are abstract, in the sense that the mechanics of the communication

get	Retrieve a data token via the port.
put	Produce a data token via the port.
hasToken(k)	Test whether *get* can be successfully applied to the port k times.
hasRoom(k)	Test whether *put* can be successfully applied to the port k times.

Fig. 8. The key communication operations in the Ptolemy II abstract semantics.

channel is not defined. It is determined by the model of computation. For instance, in synchronous dataflow[28], the channel is implemented by a queue with fixed length. In Giotto[16], the channel is a double-buffered mailbox. A value produced by a *put* operation becomes available to a corresponding *get* operation only in the next cycle of the periodic execution. In the continuous-time model of computation, the channel is a simple variable whose value is the value of a signal at the current time. A domain-polymorphic actor, like that in figure 7, is not concerned with how these operations are implemented. It is designed to the interface of the abstract semantics.

5.3. *Hierarchically Heterogeneous Composition*

A hierarchically heterogeneous model is supported by this abstract semantics as follows. Figure 4 shows an opaque composite actor. It is opaque because it contains a director. That director gives the composite a behavior like that of an atomic actor viewed from the outside. A director implements the Executable interface, and thus provides the operations of figure 6.

Suppose that in figure 4 the hierarchical abstraction of the Sinewave component is used in a model of computation different from SDF. Then from the outside, this model will appear to be a domain-polymorphic actor. When its *prefire* method is invoked, for example, the inside director must determine whether the preconditions are satisfied for the model to execute (in this case, they always are), and return true or false accordingly. When *fire* is invoked, the director must manage the execution of the inside model so that input data (if any) is read, and output data is produced. When *postfire* is invoked, the director must update the state of the model.

The communication across the hierarchical boundary will likely end up heterogeneous. In figure 4, the connection between the TrigFunction actor and the external port will be a channel obeying SDF semantics (that is, it will be realized as a finite-length queue, in this case, with length one). The connection between the external port and some other port on the outside will obey the semantics of whatever director is provided on the outside. This need not be the same as the SDF semantics.

There are many subtle issues with such hierarchical heterogeneity that are beyond the scope of this paper. In this paper, we focus on the implications for model refinement into hardware and software system realizations.

6. Actor-Oriented Model Refinement

The primary benefit of actor-oriented design is the possibility of succinctly capturing the requirements of an application by the derived properties of a model of computation. In other words, it satisfies the requirements of model-based design. As suggested in figure 3, it is distinct from program-level design, where an actor-oriented architecture may be expressed, but it is intertwinned with the specification of the actor functionality. The question remains, if we orthogonalize the actor-oriented architecture and the specification of detailed functionality, can we synthesize effective realizations?

In a purely software context, the answer is clearly yes. Simulink demonstrates this clearly with its extension Real-Time Workshop (and related products). The components in an actor-oriented architecture can be designed to an interface without a tight syntactic or semantics coupling between the component specification language and the actor assembly language.

In a hardware design context, or in hardware-software codesign, the answer is less conclusive, although there are many promising design frameworks that use this approach (see for example [44,9]). One approach is to define a platform consisting of a library of primitive actors and a model of computation for assembling them into a model. However, such a library-based approach has proved unwieldy because library development and maintenance become very difficult. Moreover, to be sufficient in all but the most domain-specific contexts, the library becomes huge, making it difficult for designers to find the components they need.

A better approach provides the designer with an actor definition language, as for example in VHDL and SystemC. In VHDL and SystemC, however, actor definition and actor assembly are bound by the syntax and semantics of the same language. Programs must be written in a highly stylized way to achieve synthesizable designs[40]. We believe that a better approach is to capture the stylized subset as a pair of syntaxes, one for actor definition and one for actor assembly. By orthogonalizing these, we get the ability to define domain-polymorphic actors, multiple models of computation, and hierarchical heterogeneity.

A still better approach is a compromise between library-based and an actor definition language. This approach is used, for example, by Real-Time Workshop, which will recognized certain primitive Simulink blocks and generate optimized code for those primitives, rather than directly using the block definition. This compromise offers the best of both approaches. We call this an *extensible library* approach.

In this section we illustrate an extensible library approach by briefly describing a prototype tool, built on Ptolemy II [8], that performs automatic refinement of synchronous dataflow specifications into both hardware and software implementations. Such a mapping is an example of a refinement relation defined above. For example, a mapping between the platform defined by the synchronous dataflow model of computation and the platform defined by synthesizable JHDL is the refinement relation $R \subset \textit{SDF models} \times \textit{JHDL programs}$. This refinement relation is represented by the descending line between *SDF models* and *JHDL programs* in figure 3. Like SystemC, our tool uses an imperative language for actor definition (Java in this case), but unlike SystemC, it orthogonalizes the actor assembly and actor definition.

Similar approaches have been demonstrated in El Greco (which became System Studio) [4], and System Canvas[35]. These all use standard imperative languages for actor definition (Java, C++ and C), but another possibility is to define a special-purpose actor definition language, such as CAL[43]. Such a language offers the compelling advantage that actor properties that are needed at the interface can be inferred rather than having to be declared.

6.1. *Model Refinement*

Synchronous dataflow has been found to describe efficient structures in both software [1],

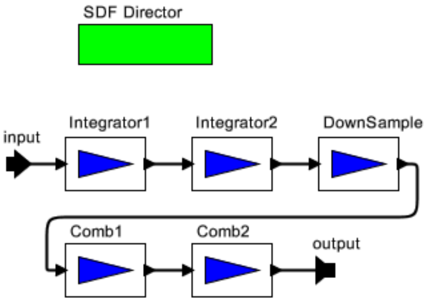

Fig. 9. A model of a two-stage CIC filter.

and hardware[44,11]. In either case, this refinement process must generate an implementation that preserves the semantics of the original model (i.e. the partial ordering of iterations and the pattern of communication), while attempting to minimize cost of implementation.

Our tool exploits the hierarchical composition of properties of the abstract semantics described above to perform hierarchical mapping of an actor-oriented model. It generates an implementation of model by composing implementations of each actor in the model according to the model of computation. The actors themselves may be specified in terms of hierarchical models, in which case the tool recursively generates a correct implementation of the hierarchical model. Or they may be specified in Java, in which case we analyze the byte code produced by a Java compiler for synthesis. Primitive actors may be recognized by the tool and synthesized directly.

6.2. *Refinement Example*

We illustrate our refinement tool using the example shown in figure 9. This model is a synchronous dataflow model, as declared by the SDF Director. This model represents a two stage *cascaded integrator-comb* or CIC filter. CIC filters are frequently used for narrow band filtering when large sample rate changes are required[18]. This model contains two instances of a discrete integrator, a downsample rate change, and two instances of a comb filter. The discrete integrator is a single-pole IIR filter with a unity feedback coefficient ($y_{\text{int}}[n] = x[n] + y[n-1]$). The downsample rate change actor decimates the signal by a factor of R and the comb-filter is an odd-symmetric FIR filter specified as $y_{\text{comb}}[n] = x[n] - x[n-R]$.

The CIC model shown in Figure 9 can be simulated for functional verification, or the model can be synthesized into one of the supported compilation targets. While it is possible to create a software executable or hardware circuit directly from the actor-oriented model of Figure 9, it is more appropriate to exploit the capabilities of existing hardware synthesis and code-generation tools provided by program-level platforms (i.e. VHDL, C++, Java, SystemC, or JHDL). To exploit these tools, the refinement process will generate a program-level design from an actor-oriented model. The final executable or circuit is then created from this program-level design using existing code-generation or hardware synthesis tools.

The CIC model, as shown, includes two specialized actors (the DiscreteIntegrator and

```
public class DiscreteIntegrator ... {
    ...

    Token _sum;    // actor state.
    Token _temp;   // temporary storage.

    public void fire() ... {
        _temp = _sum.add(input.get(0);
        output.send(0,_temp);
    }
    public boolean postfire() ... {
        _sum = _temp;
        return true;
    }
    ...
}
```

Fig. 10. An abbreviated specification of a discrete integrator.

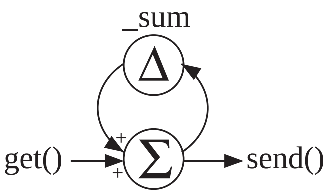

Fig. 11. Extracted SDF model of the discrete integrator.

the CombFilter) that are unlikely to be found in any but the most extensive actor libraries, so a pure library-based approach will not be sufficient. The refinement process extracts the behavior of these actors from their Java specification and synthesizes an equivalent circuit. We illustrate this process using the DiscreteIntegerator actor, shown in figure 10.

In the SDF domain, an iteration of this actor corresponds to one invocation of the fire method and one of the postfire method. The temporary storage is used to comply with the abstract semantics, where the state of the actor is not updated in the fire method. If this actor is to be used only in SDF, then this policy is not necessary. However, by following it, we define a domain-polymorphic actor that can be used, for example, in a synchronous/reactive model of computation, where the semantics is based on iteration to a fixed point.

The invocations of the get and send methods in fire correspond to the communication operations in figure 8. In each iteration, this actor obtains a data token at its input port, adds the value of the data token to the internal `_sum` variable (using a polymorphic add method), and sends the resulting sum to other actors through its output port. Before completing the iteration, the state of the `_sum` variable is updated in the postfire method. Our tool analyzes the Java byte code produced by compiling this definition and extracts this behavior into the data flow graph in figure 11.

A similar process occurs for the CombFilter actor, with the additional complication that

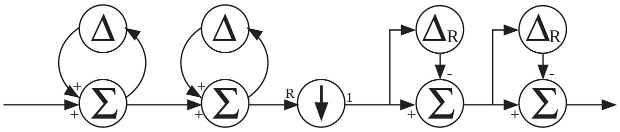

Fig. 12. Flat SDF model of CIC decimator.

a bulk delay must be recognized in the Java byte code in order to do effective synthesis. The Downsample actor, on the other hand, represents a primitive rate change operation and is treated as a library actor. The data flow graph that is used for synthesis is shown in figure 12.

7. Conclusion

A platform is a set of designs. Model-based design is about using platforms with useful modeling properties to specify designs, and then synthesizing implementations from these specifications. A design language, such as SystemC or VHDL, defines a platform. Any valid expression in the language is an element of the set. Viewed from below, the language provides an abstraction of implementation capabilities. Viewed from above, the language provides a set of constraints together with benefits that flow from those constraints. It provides a conceptual framework within which a design is crafted.

An actor-oriented platform lies above the program-level platforms that are widely used today. It orthogonalizes the actor definition language and the actor assembly language, enabling domain-polymorphic actor definitions, multiple models of computation, and hierarchical heterogeneity. Actor-oriented platforms offer compelling modeling properties. Synthesis into program-level descriptions is possible, although much work remains to be done to make it fully effective.

8. References

1. S. S. Battacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.
2. A. Benveniste and G. Berry. The synchronous approach to reactive and realtime systems. *Proceedings of the IEEE*, 79(9):1270–1282, Sept. 1991.
3. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.
4. J. Buck and R. Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES), San Diego, California*, May 2000.
5. J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, Electrical Engineering and Computer Sciences, University of California Berkeley, 1993.
6. J. R. Burch, R. Passerone, and A. Sangiovanni-Vincentelli. Using multiple levels of abstractions in embedded software design. In T. A. Henzinger and C. M. Kirsch, editors, *Proceedings of EMSOFT 01: Embedded Software*, Lecture Notes in Computer Science 2211, pages 166–184. Springer, 2001.

7. J. Cockx. Requirements for software modeling in SystemC 3.0. Posted to the SystemC Forum, Mar. 2002.
8. J. Davis et al. Ptolemy II - Heterogeneous concurrent modeling and design in Java. Memo M01/12, UCB/ERL, EECS UC Berkeley, CA 94720, Mar. 2001.
9. W. R. Davis. *A hierarchical, automated design flow for low-power, high-throughput digital signal processing ICs*. PhD thesis, UC Berkeley, 2002.
10. E. de Kock, G. Essink, W. Smits, P. van der Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. Yapi: Application modeling for signal processing systems. In *Proceedings of the 37th Design Automation Conference (DAC'2000)*, pages 402–405, June 2000.
11. M. Edwards and P. Green. The implementation of synchronous dataflow graphs using reconfigurable hardware. In *Proceedings of Field Programmable Logic Symposium*, volume 1896 of *Lecture Notes in Computer Science*, pages 739–748. Springer, 2000.
12. K. Furuta, M. Yamakita, and S. Kobayashi. Swingup control of inverted pendulum using pseudo-state feedback. *Journal of Systems and Control Engineering*, 206:263–269, 1992.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
14. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1321, September 1991.
15. C. Hansen. Hardware logic simulation by compilation. In *Proceedings of the Design Automation Conference (DAC)*. SIGDA, ACM, 1988.
16. T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. In T. Henzinger and C. Kirsch, editors, *Proceedings of EMSOFT 01: Embedded Software*, Lecture Notes in Computer Science 2211, pages 166–184. Springer-Verlag, 2001.
17. T. A. Henzinger and C. M. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proceedings of Conference on Programming Language Design and Implementation(PLDI)*. SIGPLAN, ACM, June 2002.
18. E. B. Hogenauer. An economical class of digital filters for decimation and interpolation. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 29(2):155–162, 1981.
19. B. Hutchings et al. A CAD suite for high-performance FPGA design. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–24. IEEE Computer Society Press, 1999.
20. G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475, Paris, France, 1974. International Federation for Information Processing, North-Holland Publishing Company.
21. G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of the IFIP Congress 77*, pages 993–998, Paris, France, 1977. International Federation for Information Processing, North-Holland Publishing Company.
22. K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonolization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), Dec. 2000.
23. G. Kiczales et al. Aspect-oriented programming. In *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, number 1241 in Lecture Notes in Computer Science, pages 220–242. Springer-Verlag, 1997.
24. T. J. Koo, J. Liebman, C. Ma, and S. Sastry. Hierarchical approach for design of multi-vehicle multi-modal embedded software. In T. Henzinger and C. Kirsch, editors, *Proceedings of EMSOFT 01: Embedded Software*, Lecture Notes in Computer Science

2211, pages 344–360. Springer-Verlag, 2001.

25. H. Kopetz and G. Grunsteidl. TTP – a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27:14–23, Jan. 1994.
26. P. le Guernic, A. Benveniste, P. Bournai, and T. Gauthier. Signal: A data flow oriented language for signal processing. Technical report, IRISA, Rennes France, 1985.
27. A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. T. IV, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Proceedings of Workshop on Intelligent Signal Processing*, May 2001.
28. E. Lee and D. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, pages 55–64, September 1987.
29. E. A. Lee and S. Neuendorffer. MoML - a modeling markup language in XML Version 0.4. Technical Memorandum UCB/ERL M01/12, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, CA 94720, USA, March 2000.
30. S. Y. Liao. Towards a new standard for system-level design. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*. SIGDA, ACM, May 2000.
31. S. Y. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceedings of the 34th Design Automation Conference (DAC'1997)*. SIGDA, ACM, 1997.
32. J. Liu. *Responsible Frameworks for Heterogenous Modeling and Design of Embedded Systems*. PhD thesis, EECS Department, University of California at Berkeley, CA, 2001.
33. J. Liu, J. Eker, J. W. Janneck, and E. A. Lee. Realistic simulations of embedded control systems. In *Proceedings of the International Federation of Automatic Control 15th World Congress*, July 2002.
34. J. Ludvig, J. McCarthy, S. Neuendorffer, and S. R. Sachs. Reprogrammable platforms for high-speed data acquisition. In *Proceedings of the 35th Asilomar Conference on Signals, Systems, and Computers*, Nov. 2001. Revised version to appear in Journal of Design Automation for Embedded Systems.
35. P. K. Murthy, E. G. Cohen, and S. Rowland. System Canvas: A new design environment for embedded DSP and telecommunication systems. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*. SIGDA, ACM, Apr. 2001.
36. T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Department, University of California at Berkeley, CA, 1995.
37. J. A. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the Design Automation Conference (DAC)*, June 1997.
38. A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign*, Feb. 2002.
39. S. Swan. An introduction to system level modeling in SystemC 2.0. Technical report, Open SystemC Initiative, May 2001.
40. Synopsys. *Describing Synthesizable RTL in SystemC*, Jan. 2002. Available at http://www.synopsys.com.
41. J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, pages 110–112, Apr. 1997.
42. J. Teich, E. Zitzler, and S. Bhattacharyya. 3D exploration of software schedules for DSP algorithms. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*. SIGDA, ACM, May 1999.
43. L. Wernli. Design and implementation of a code generator for the CAL actor language. Technical Memorandum UCB/ERL M02/5, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, CA

94720, USA, March 2002.
44. M. Williamson. *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications*. PhD thesis, UC Berkeley, 1998.