

Taming Heterogeneity—the Ptolemy Approach

Johan Eker^{*}, Jörn W. Janneck^{*}, Edward A. Lee^{*}, Jie Liu^{*}, Xiaojun Liu^{*}, Jozsef Ludvig[†],
Stephen Neuendorffer^{*}, Sonia Sachs^{*}, Yuhong Xiong^{*}

^{*}EECS Department, University of California at Berkeley, Berkeley, CA
{johane,janneck,eal,liuj,liuxj,neuendor,ssachs,yuhong}@eecs.berkeley.edu

[†]Athena Semiconductors Inc., 47358 Fremont Blvd., Fremont, CA 94538
jludvig@directvinternet.com

to appear in Proceedings of the IEEE, 2002

Abstract—Modern embedded computing systems tend to be *heterogeneous* in the sense of being composed of subsystems with very different characteristics, which communicate and interact in a variety of ways—synchronous or asynchronous, buffered or unbuffered, etc. Obviously, when designing such systems, a modeling language needs to reflect this heterogeneity. Today’s modeling environments usually offer a variant of what we call *amorphous heterogeneity* to address this problem. This paper argues that modeling systems in this manner leads to unexpected and hard-to-analyze interactions between the communication mechanisms and proposes a more structured approach to heterogeneity, called *hierarchical heterogeneity* to solve this problem. It proposes a model structure and semantic framework that support this form of heterogeneity, and discusses the issues arising from heterogeneous component interaction and the desire for component reuse. It introduces the notion of *domain polymorphism* as a way to address these issues.

I. INTRODUCTION

Modern embedded systems are typically software-enabled, interconnected, and engage the physical world. Among other things, they must be reliable, concurrent, and reactive in real-time. To ensure these requirements in the face of growing complexity, designers may be able to use formal methods that allow designs to be analyzed and verified. Embedded systems also tend to be *heterogeneous*, i.e. they include subsystems with very different characteristics such as mechanical, hydraulic, analog, and digital hardware, as well as software that is oriented towards data flow, realizes control logic, deals with resource allocation, or must provide some real-time performance. Much effort has been invested in creating formal models that elegantly describe each of these characteristics, and the selection of formal models and tools when designing an embedded system involves decisions that can have important consequences for a project.

However, each model typically only represents one aspect of the entire system, and thus only part of its total behavior. In order to evaluate the behavior of the system as a whole, these models must be composed in some fashion so that their properties can be considered together. This *heterogeneous composition* must represent interaction and communication be-

tween models while preserving the properties of each individual model.

Brute-force composition of heterogeneous models may cause what we call *emergent behavior*. Model behavior is emergent if it is caused by the interaction between characteristics of different formal models, and if it was not intended or foreseen by the author of the model. Emergent behavior is therefore always a surprise to the designer, and potentially violates properties the designer expects from the individual formal models, thereby interfering with the ability to analyze the entire model.

One example of an emergent behavior is priority inversion between threads in a real-time operating system. In this case threads are interacting using two different communication mechanisms: mutual exclusion using monitors and a fixed priority scheduling mechanism. Looking at each mechanism in isolation, a designer naturally expects that the thread scheduler will preempt low priority threads when a high-priority thread is ready to execute. Instead, by locking a monitor, a low priority thread may stall a high priority thread for an unbounded amount of time.

We will call a style of handling heterogeneous models that allows various interaction mechanisms to be specified between a group of components at the same time *amorphous*, since it does not constrain the applicability of different interaction styles by scoping or similar constructions. Constructing complex models in an environment that supports amorphous heterogeneity is prone to produce models that exhibit emergent behavior such as the priority inversion described above.

This work describes a more structured, *hierarchically heterogeneous*, approach to compose different models. This approach is studied in the Ptolemy project and implemented in the Ptolemy II software environment. Using hierarchy, one can effectively divide a complex model into a tree of nested sub-models, which are at each level composed to form a network of interacting components. Our approach constrains each of these networks to be locally homogeneous, while allowing different interaction mechanisms to be specified at different levels in the hierarchy.

One key concept of hierarchical heterogeneity is that the interaction specified for a specific local network of components

covers both the flow of data as well as the flow of control between them. We call such a framework a *model of computation* (MoC), as it defines how computation takes place among a structure of computational components, effectively giving a semantics to such a structure.

In order to facilitate hierarchical composition, a MoC must be compositional in the sense that it not only aggregates a network of components, but also turns that network itself into a component that in turn may be aggregated with other models, possibly under a different MoC. We present an automata-based approach to study the compatibility and compositionality of components with their MoC frameworks.

Because each local submodel is governed by one well-defined MoC, it can be understood solely in the terms of this MoC, and it can be analyzed in the formal framework defined by it, while at the next higher level of the hierarchy the submodel is considered atomic for the purpose of understanding or analyzing its containing model. This localization of analyzability allows individual components to be refined into more detailed models without affecting the properties of the rest of the system. Thus, analysis, compilation and code generation become compositional.

The remainder of this paper is structured as follows. Section II summarizes some of the related work in this area. Section III motivates hierarchical decomposition with a small example, while section IV introduces the basic structure of these models and the notion of a *domain* which implements a formal model of computation. Section V describes some of the issues that arise in hierarchical heterogeneous modeling, especially concerning which models can be embedded in which other models, and discusses our approach. Section VI sketches the contributions of a hierarchically heterogeneous model structure to the task of generating code for a variety of targets, and we conclude in section VIII.

II. RELATED WORK

Different modeling techniques reflect different ways of thinking by designers, and their intentions when abstracting system properties.

Continuous-time models, such as ordinary differential equations and differential algebraic equations, have been used for modeling mechanical dynamics, analog circuits, chemical processes, and many other physical systems. Design tools such as Spice [40], Saber (by Analogy Inc.), and early versions of Simulink (by The MathWorks) are based on these models.

Discrete-event models have a global notion of time and time-stamped events. They are suitable for modeling timing properties in digital circuits, network traffic, and queuing systems. Languages and tools like VHDL [41], Verilog [45], and ns [10] are primarily based on these models. In some discrete-event models, the time stamps of all events are multiples of a pre-defined time interval. These discrete-time models are used in discrete control systems and cycle-accurate simulations.

In many system modeling methodologies, time is abstracted away. Synchronous/reactive (SR) models [6] abstract time into discrete “ticks”, and all events in the system are synchronized to them. A system reacts to these events, and the reaction is assumed to take zero time – the synchrony assumption. Modeling

methodologies like Statecharts [21] and languages like Esterel [7], Signal [19], Lustre [20], and Argos [36] realize examples of these models.

Software systems usually have an even higher level of abstraction, such that the ordering relation only exists among a subset of events in the system. Synchronous message passing models, like Hoare’s communicating sequential processes (CSP) [23] and Milner’s calculus of communicating systems (CCS) [37], use an atomic message exchange mechanism to model communications among processes. They clearly specify communication events among processes, but not the interleaving of events inside parallel processes. Examples of languages that support this model of computation are Lotos [11], and Occam [47].

Asynchronous message passing models of computation, such as Kahn’s process networks [24] and dataflow models [31], use FIFO queues to model the communication among processes. Only events within the same communication channel are strictly ordered, while the ordering among events in different channels remains unspecified. Various special cases of dataflow models, like synchronous dataflow (SDF) [30], Boolean dataflow [13] and cyclo-static dataflow [28] are used for specifying signal process algorithms, since in these models, many properties like deadlock and memory requirements are statically analyzable. Dataflow models have been used within tools such as GRAPE II [28], SPW (by Cadence) and COSSAP [27].

The above models usually characterize only a single aspect of a complex system. For heterogeneous systems, as seen in communication systems, electro-mechanical or electro-optical systems, software enabled control systems, and hybrid systems, the modeling and design methodologies and environments that can integrate semantically distinct models are in high demand [18] [39] [5].

Many languages and tools, that were developed based on a single model, start to embrace other models. VHDL-AMS [16] extends the discrete-event based VHDL language with the ability to model and simulate analog components. Recent versions of Simulink [38] have the capabilities of integrating discrete-time and state-transition components with a continuous-time framework. Extensions have been made to the Esterel language to include asynchronous computations [42], [1].

New frameworks have also been developed to formally characterize heterogeneity in systems. The hybrid system theories integrate continuous-time differential equations with discrete automata [4]. The *charts (pronounced: star-charts) formalism combines finite state machines with a variety of concurrent models of computation [29]. New languages and tools emerge to support heterogeneous modeling approaches. The Moses framework [17] is being developed to support various event-based models, like Petri nets, data flow and discrete-event models. The El Greco system [12] (now called SystemStudio) supports a generalized form of the cyclo-static dataflow (CSDF) model of computation [9] and allows hierarchical combination of state machine-based control logic and dataflow models. A language under development, Rosetta [3], provides support for specifying multiple aspects of a system, each one belonging to a different domain. However, most of these languages and tools are based on a limited number of models and restrict the way

that they can be combined.

The Ptolemy II software environment [14] provides support for hierarchically combining a large variety of models of computation and allows hierarchical nesting of the models. Modeling, simulation, and design of large, concurrent, real-time systems can be achieved with a high level of confidence that emergent behavior will not occur.

III. MOTIVATING EXAMPLE

In this section we try to illustrate some of the issues arising in modeling a heterogeneous system by walking through a small example. We represent the models as block diagrams, since this seems to be the most readily understandable representation, and because block diagrams are a very common way to present artifacts in science and engineering in general, and in Ptolemy in particular.

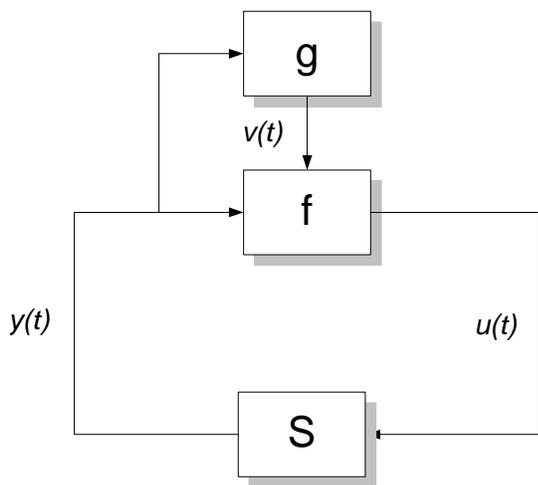


Fig. 1. A model of a continuous-time system.

The model in Fig. 1 is intended to represent a system consisting of three *components* (represented by blocks) f , g , and S , connected by continuous *signals* $u(t)$, $y(t)$ and $v(t)$ (represented by the arcs between the blocks). These signals have a direction, i.e. each is created by one of the components and received by one or several components. Since they are continuous, we may think of them as having a well-defined value for each point t in time, as they would e.g. if they were electrical signals. This is a typical model of continuous system with a process S , governed by a controller consisting of two sub components f and g .

Abstractly, the arcs then represent functions from the real numbers (or some appropriately chosen interval in \mathbb{R}) to some set of values, while the blocks stand for functions on these continuous functions which can be described by equations. For example, the effect of the system S may be written as

$$y(t) = (S(u))(t) \quad \text{with } t \in \mathbb{R}.$$

Similarly, the two controller blocks may be construed as equations between their input signals and output signals, i.e.

$$\begin{aligned} v(t) &= (g(y))(t) \\ u(t) &= (f(v, y))(t) \end{aligned}$$

which can be transformed into

$$u(t) = (f(g(y), y))(t).$$

In the context of this discussion, it is important to note that the model in Fig. 1 can be readily understood as unambiguously describing a system, and we can easily derive a formal mathematical description from it, as well as an understanding of what it means in terms of a specific application.

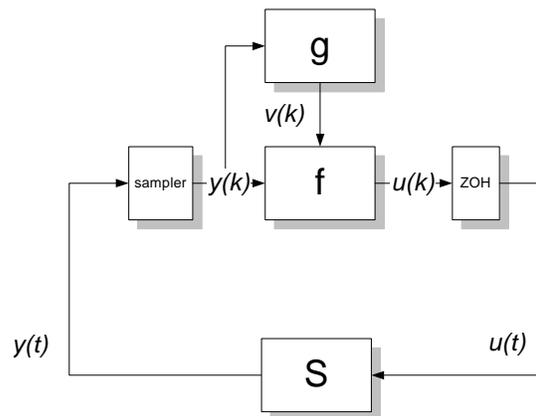


Fig. 2. An amorphous model of a heterogeneous system.

Now we modify this model slightly by making the controller *discrete* in time, executing periodically at a given frequency. The controller interfaces the continuous-time system S by a sampler (which transforms the continuous signal $y(t)$ into a discrete sequence of values $y(k)$) and a zero-order hold (which conversely maps a sequence of values $y(k)$ into a continuous-time signal $y(t)$), as shown in Fig. 2.

The result of this is in fact a *heterogeneous* system, consisting of continuous and discrete components. The model now contains more than one kind of interaction between components—in addition to the continuous-time signals from the previous example, we also have discrete signals between the components describing the controller.

Interpreting this model, however, is not quite as straightforward as it was for the purely continuous model, because there are several possible interpretations of what this model does, and they differ in the way they order the computations performed in the blocks f and g . When a new sample is generated, the new value, say $y(k)$, is available to both, f and g . If now g is executed before f , it computes the value $v(k) = (g(y))(k)$, which is sent to f . If now f executes, it takes this value as the current one, and computes $u(k) = (f(v, y))(k)$.

However, if first f executes, it has $y(k)$ and the previous value produced by g as inputs, and thus produces $u(k) = (f(\Delta(v), y))(k)$, where Δ is a unit delay, i.e. $(\Delta(v))(k) = v(k-1)$, for any integer k . This is then followed by the execution of g , computing $v(k) = (g(y))(k)$, although this value will not be used by f until the next sample.

The key to this ambiguity is that while the continuous-time portions of the model have a cleanly defined, Newtonian physical notion of time, the discrete portions have not been assigned a clear semantics. What is the value of v at time t ? The function v is indexed by an integer. Suppose we declare that at time t

has value $v(k)$ if $kT \leq t < (k+1)T$. Then we get the first execution. On the other hand, if we declare that v has value $v(k)$ if $kT < t \leq (k+1)T$, then we get the second. These are equally valid ways to reconcile the disparate semantics of continuous and discrete time.

If we take the second interpretation, then we should perhaps be consistent and choose a similar interpretation for the discrete signal y . This would introduce a one-step delay between the sampler and f . Without this, however, we would have a weirdly amorphously heterogeneous model.

The choice has implications for implementation. For instance, it is relatively easy to assume a specific kind of communication between components (say, discrete signals are sent asynchronously between components as *tokens*), and describe it and a suitable strategy for the flow of control among them. Such a description could say that a data token, after it has been received and then used in some computation, is consumed, and the component cannot do any more computation until it has a 'fresh' token on each of its input lines. This rule would disambiguate the above case, as f could not compute until g has finished and produced a fresh result token for $v(k)$.

On the other hand, this rule assumes that the components we apply it to are engaged in only one kind of data exchange—if we imagine f to produce, say, a continuous output signal, we would need another rule to tell us which value this signal has during the time that our first rule says that f may not compute. And so on for each new kind of signal we care to connect to f .

So the desire for a precise and unambiguous description of the data flow and control flow between components, which works best in homogeneous models that have only one kind of data flow between components, seems to oppose the use of heterogeneous models. Clearly, arbitrarily mixing different kinds of data flow between components, as in Fig. 2, in an amorphously heterogeneous way, leads to ambiguities which are usually much harder to locate than in our trivial example above.

The Ptolemy approach, on the other hand, reconciles the wish for a homogeneous and thus predictable model with the desire to mix partial models of different kinds in a common heterogeneous model by hierarchically nesting sub-models of potentially different kinds.

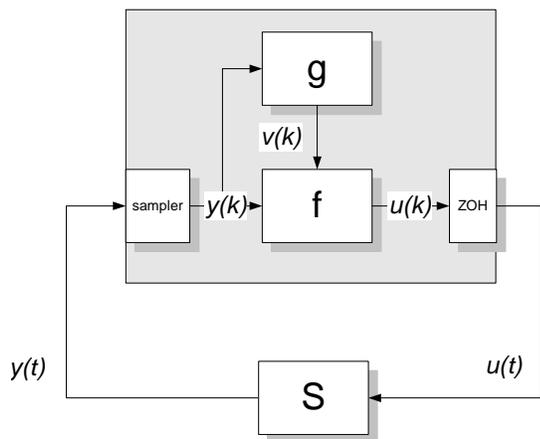


Fig. 3. A hierarchically heterogeneous version of the model in Fig. 2.

The model in Fig. 3 shows how the amorphous model from Fig. 2 is disambiguated by the use of hierarchical heterogeneity. The controller components f and g are now contained in a submodel which is embedded in a (continuous-time) top-level model. Both the top-level model and the controller submodel are homogeneous, and are governed by a specific set of rules that determine how data flows and when and how computation happens, in this case, a continuous-time top-level model, and a synchronous dataflow model for the discrete controller model.

In fact, the rules for synchronous data flow models say that a component consumes tokens upon performing a computational step, and requires a predefined number of tokens to be present (here always 1) before being able to make this step. By placing or not placing an initial token onto the $v(k)$ signal between g and f , we can even select unambiguously one of the two possible behaviors of the discrete part of this model—as shown in Fig. 4.

Synchronous dataflow does not have a notion of time, and it does not need one in order to interact with continuous time models. Reactions of the synchronous subsystem are instantaneous from the perspective of the continuous system.

IV. MODEL STRUCTURE AND SEMANTICS

A. Actor-oriented modeling and design

Ptolemy advocates an *actor-oriented* view of a system, where the basic building block of a system is an *actor*. Actors are concurrent components that communicate via interfaces called *ports*. Relations define the interconnect between these ports, and thus the communication structure between actors.

Actor orientation is a way of structuring and conceptualizing a system that complements the object-oriented techniques that today are widely used to structure software systems. Viewing a system as a structure of actors emphasizes its causal structure and its concurrent activities, along with their communication and data dependencies. An object-oriented perspective sees the state of a system as a structure of objects that are related by references to each other. A consequence of an actor-oriented view of a system is a decoupling of the transmission of data from the transfer of control—by contrast, object-orientation has come to mean that objects can only communicate by *calling* each other's methods, effectively transferring control to each other's code.

By contrast, the communication among actors may or may not be related to the flow of control. Ptolemy actors generalize those described by Agha [2] in that they are not necessarily associated with a thread of control. Each actor may run in its own thread, or the entire model may run sequentially in a single thread. It is the model of computation, rather than the individual actors themselves, that defines the details of scheduling and communication, and whether and how these are related.

Factoring this functionality out of the actors into a common coordinating context has two important consequences. First, it makes actors much more reusable, as an actor describes abstract functionality that can be run in a large number of different ways. Secondly, a composition of actors governed by some model of computation can more easily be analyzed and understood, as there is no way that different actors might have made different

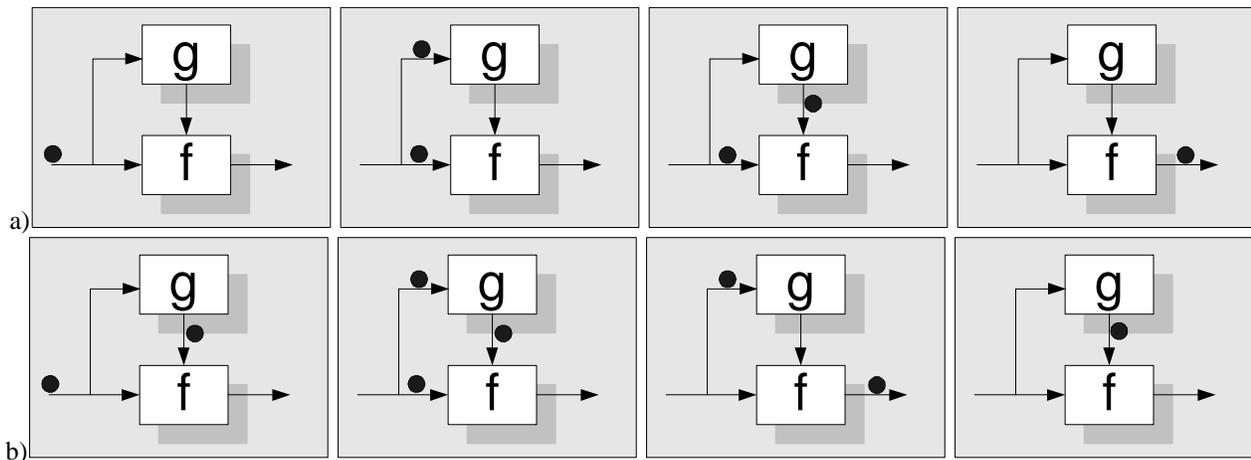


Fig. 4. The two different behaviors for the discrete controller.

assumptions about the way control flow and communication are handled.¹

The structure imposed by a model of computation on a collection of actors may be exploited in a number of ways. The system may now possibly be analyzed for certain properties (such as absence of deadlocks, boundedness of resources, fairness of execution, timeliness of results), and of course compilers translating a model to a target architecture might also use the knowledge about the model of computation to optimize the generated code (e.g. by computing static schedules, using buffers efficiently or removing redundant checks for conditions that are guaranteed by the model of computation).

We will now present the structure of a Ptolemy model in more detail, showcase some common models of computation used in embedded systems modeling, and discuss the issues arising from hierarchical composition of models of computation.

B. Model structure

An actor can be *atomic*, in which case it must be at the bottom of the hierarchy. An actor can be *composite*, in which case it contains other actors. Note that a composite actor can be contained by another composite actor, so hierarchies can be arbitrarily nested.

Actors have *ports*, which are their communication interfaces. A port can be an input port, an output port, or both. Communication channels are established by connecting ports. A port for a composite actor can have both connections to the inside and to the outside, thus link the communication from inside actors to outside actors.

For example, in Figure 5, the top-level composite actor contains actors A1 and A2. Actor A2 is also a composite actor, as suggested by the different icon, which contains actors A3 and A4. Actors A1, A3, and A4 are atomic actors. Port P2 of A2 is an external port and connects to port P1 on the outside and port P3 on the inside. Note that the top-level composite actor has no external ports, implying that it completely encapsulates the design.

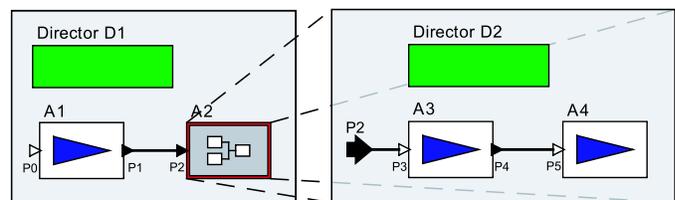


Fig. 5. A hierarchical model in Ptolemy II.

C. Execution

Actors, both atomic and composite, are executable. The execution of a composite actor is a controlled composition of the execution of the actors it contains. This compositionality of execution is the key to managing heterogeneity hierarchically.

An execution in Ptolemy II has the following phases: *setup*, *iterate*, and *wrapup*, where each phase may have more fine-grained phases.

The setup phase is divided into *preinitialize* and *initialize* phases. The preinitialization subphase usually deals with structural information, like constructing dynamically created actors, determining the widths of ports, and creating (domain specific) receivers. The initialization phase initializes parameters, resets local state, and produces initial tokens. Typically, the preinitialization of an actor is performed exactly once during the actor's life of execution, and before any other execution of the actor. The initialization of an actor is performed once after the preinitialization and type resolution. It may be performed again if the semantics requires that the actor to be re-initialized, for example, in the hybrid system formalism [33].

Actors perform atomic executions (called *iterations*) in the iterate phase. An iteration is a finite computation that leads the actor to a quiescent state. The model of computation of a composite actor determines how the iteration of one actor is related to the iterations of other actors in the same composite.

In order to coordinate the iterations among actors, an iteration is further broken down into prefire, fire, and postfire. Prefire tests the preconditions for the actor to execute, such as

¹Of course, not all problems concerning actor compatibility simply disappear,

but as we will see below, this framework provides promising approaches for addressing these issues.

the presence of sufficient inputs to complete the iteration. The computation of the actor is typically performed during the fire phase. Typically, computation involves reading inputs, processing data, and producing outputs. The persistent state of the actor is updated in postfire.

Note that despite the fact that computation occurs during the fire phase, the state of the actor is not updated until postfire. This supports fixed-point iteration in some models of computation, such as synchronous reactive models and continuous time differential equations. These models of computation compute the fixed-point of actor outputs while keeping the state of each actor constant. The state of an actor can only be updated after the fixed-point has been reached. This requires the firing of each actor several times before the actor is postfired.

The execution is wrapped up exactly once at the end of the execution. Typically, actors release resources that were allocated during execution.

D. Domains

In Ptolemy II an implementation of a model of computation associated with a composite actor is called a *domain*.² A domain defines the communication semantics and the execution order among actors. It is realized by two classes, a *director* and a *receiver* class.

The communication mechanism is implemented using *receivers*. Receivers are contained in input ports, and there is one receiver for each communication channel. Receivers could represent FIFO queues, mailboxes, proxies for a global queue, or rendezvous points. In the model in Figure 5, there is a receiver in the input ports P2, P3, and P5. Port P0 has no receiver because it is not connected to an input channel.

Actors, when resident in a domain, acquire domain-specific receivers. By separating computation and communication in this way, many actors (called *domain-polymorphic actors*) can be reused in different domains. We discuss domain-polymorphism in further detail in Section V.

The execution order of the actors contained in a composite is controlled by a *director*. Since receivers and directors must work together, the director is also responsible for creating receivers. When a composite actor is fired, the director inside the composite actor fires the actors of the contained model.

Figure 5 shows a hierarchical model using two different domains. In the model, the top-level composite actor contains a director D1 and A2 is a hierarchical composite actor with director D2. Hence, director D1 controls the execution order of actors A1 and A2 and director D2 controls the execution of A3 and A4 whenever A2 is executed. The receiver in port P1, created by Director D1, mediate communication between ports P1 and P2. Likewise, receivers created by director D2 mediate communication between ports P2 and P3, and between ports P4 and P5.

We present here some domains that we have realized in Ptolemy II. This is far from a complete list. The intent is to show the diversity of the models of computation under study.

²The term “domain” comes from a fanciful notion in astrophysics, that there are regions of the universe with different sets of laws of physics. A model of computation represents the “laws of physics” of the submodel governed by it.

1) *Communicating Sequential Processes*: In the communicating sequential processes (CSP) domain, created by Neil Smyth, actors represent processes that communicate by atomic instantaneous rendezvous. Receivers in this domain implement the rendezvous points. An attempt to put a token into a receiver will not complete until a corresponding attempt is made to get a token from the same receiver, and vice versa. As a consequence, the process that first reaches a rendezvous point will stall until the other process also reaches the same rendezvous point [22].

2) *Continuous time*: The continuous time (CT) domain [34] models ordinary differential equations (ODEs), extended to allow the handling of discrete events. Special actors that represent *integrators* are connected in feedback loops in order to represent the ODEs. Each connection in this domain represents a continuous-time function, and the components denote the relations between these functions.

Each receiver in the CT domain is a buffer with size one. It contains the value of the continuous function of the corresponding connection at a specific time instant. The execution of a CT model involves the computation of a numerical solution to the ODEs. In an iteration of a CT model, time is advanced by a certain amount and a fixed-point value of all the continuous functions is computed. As mentioned previously, this fixed-point computation may involve firing individual actors multiple times.

3) *Discrete event*: In the discrete event (DE) domain, created by Lukito Muliadi, actors communicate through events placed on a (continuous) time line. Each event has a value and a time stamp. Actors process events in chronological order. The output events produced by an actor are required to be no earlier in time than the input events that were consumed. In other words, DE models are *causal*.

The execution of this model uses a global event queue. When an actor generates an output event, the event is placed in the queue, and sorted according to its time stamp. Receivers in this domain are proxies for the global event queue. During each iteration of a DE model, the events with the smallest time stamp are removed from the global event queue, and their destination actor is fired.

4) *Process network*: In the process network (PN) domain, created by Tom Parks, actors represent processes that communicate by (conceptually infinite capacity) FIFO queues [25] Receivers in this model implement these FIFO queues. Writing to the queues always succeeds, while reading from an empty queue blocks the reader process. The simple blocking-read-nonblocking-write rule ensures the determinacy of the model.

5) *Synchronous dataflow*: A synchronous dataflow (SDF) model [30] is a particularly restricted special case of a PN model. When an actor is executed in this model, it consumes a fixed number of tokens from each input port, and produces a fixed number of tokens to each output port. A valuable property of SDF models is that deadlock and boundedness can be statically analyzed. Receivers in this domain represent FIFO queues with fixed finite capacity, and the execution order of components is statically scheduled.

V. COMPONENT INTERACTIONS AND DOMAIN POLYMORPHISM

A domain controls the communication between actors³ by determining the receivers on their input ports. Also, the director of a domain may control the execution order of the actors. In effect, the receiver and director decouples the idiosyncrasies of interaction in a domain from the requirements of the actors embedded in it.

The concept of domain-specific receivers and directors that form the interface between actors and their environment seems to suggest that any actor, including every model in any domain, may be embedded into any other model, because all rely on a common notion of execution (atomic firing) and a common notion of communication (receivers). Clearly, this is most desirable, since it maximizes the reuse of components and models. Unfortunately, things are not so simple.

In software terms, by defining the receiver notion we have essentially specified an *interface*, a set of *lexical* conventions for sending or receiving information to and from the environment of an actor. For example, all receivers have a `get()` method, which is used to obtain a token stored in the receiver. Since the receiver notion is generic, there are many different realizations of these conventions (which is, after all, the whole point in the design of this abstraction). Unfortunately, any two such realizations are not necessarily compatible.

For instance, all the actors have a `fire()` method that the director uses to start their execution. In this method, an actor may call the `get()` method of the receiver to obtain an input token. In the synchronous data flow (SDF) domain, the scheduler guarantees that an actor is fired only when there is a token in its receiver. So an SDF actor does not need to check the availability of token before calling `get()`. By contrast, in the discrete event (DE) domain, the director does not make the same guarantee, so the actors should check the availability of a token by calling `hasToken()` before calling `get()`. If a SDF actor is used in DE, it may cause an exception by calling `get()` on an empty receiver.

To ensure validity of our models, however, we would like to statically check the compatibility of an actor with a domain it is embedded in. This suggests that we need to precisely specify the behavior of the receivers and directors in different domains. Our approach is to define automata that model the combined role of receivers and directors. These automata are called domain automata since they essentially specify the communication behavior of the domains. We also describe the behavior of the actors using automata and check the compatibility of the actor with a domain by composing the domain automata with the actor automata. This approach is sketched below. More information can be found in [32].

Among the many variants of automata models, we choose interface automata [15] for their strength in the composition semantics. As with other automata models, interface automata consist of states and transitions⁴ and are usually depicted by

³Remember that these actors may be atomic or may, in fact, themselves be formulated as models in some, possibly different, domain. It is testimony to the power of the domain framework that this distinction is irrelevant to the following discussion.

⁴Transitions are called actions in [15].

bubble-and-arc diagrams. There are three different kinds of transitions in interface automata: input, output, and internal transitions. When modeling a software component, input transitions correspond to the invocation of methods on the component, or the returning of method calls from other components. Output transitions correspond to the invocation of methods on other components, or the returning of method calls from the component being modeled. Internal transitions correspond to computations inside the component. For example, Fig. 6 shows the interface automaton model for an SDF actor. This figure is a screen shot of a model in the Ptolemy II interface automata domain. The convention in interface automata is to label the input transitions with an ending “?”, the output transitions with an ending “!”, and internal transitions with an ending “;”. Fig. 6 does not contain any internal transitions. The block arrows on the sides denote the inputs and outputs of the automaton. They are:

- f: the invocation of the `fire()` method of the actor.
- fR: the return from the `fire()` method.
- g: the invocation of the `get()` method of the receiver at the input port of the actor.
- t: the token returned by the `get()` call.
- hT: the invocation of the `hasToken()` method of the receiver.
- hTT: the value true returned by the `hasToken()` call, meaning that the receiver contains one or more tokens.
- hTF: the value false returned by the `hasToken()` call, meaning that the receiver does not contain any token.

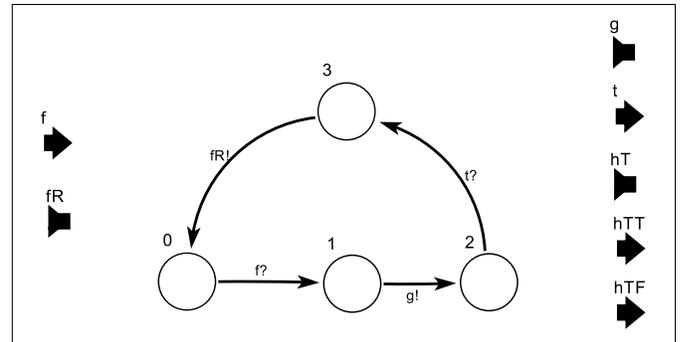


Fig. 6. Interface automaton model for a SDF actor.

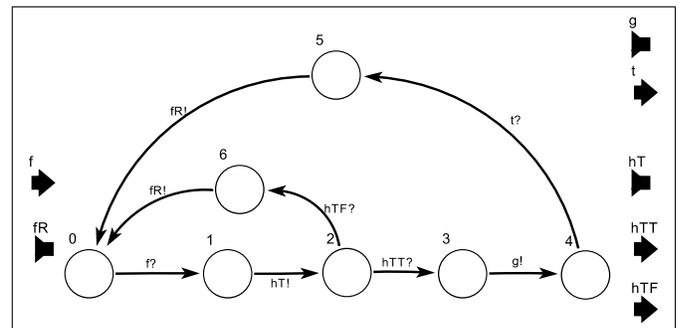


Fig. 7. Interface automaton model for a domain-polymorphic actor.

The initial state is state 0. When the actor is in this state, and the `fire()` method is called, it calls `get()` on the receiver to obtain a token. After receiving the token in state 3, it performs

some computation, and returns from `fire()`. Note that this actor does not check the availability of tokens before calling `get()`. By contrast, the automaton in Fig. 7 describes an actor that performs this check. We will show that this latter actor is domain polymorphic in that it can work in multiple domains.

Fig. 8 and Fig. 9 show the domain automata for the SDF and DE domains, respectively. Here, p and pR represent the call and the return of the `put()` method of the receiver, which is used by an actor to put a token into the receiver. The SDFDomain automaton encodes the assumption of the SDF domain that an actor is fired only after a token is put into the receiver. On the other hand in the DE domain, an actor may be fired without a token being put into the receiver at its input. This is indicated by the transition from state 0 to state 7 in Fig. 9.

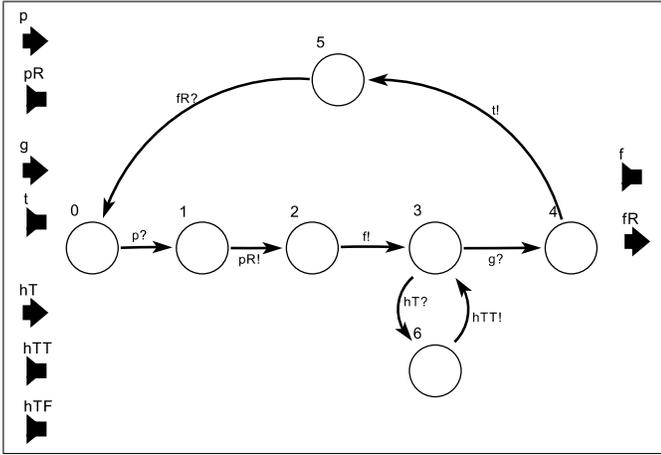


Fig. 8. SDF Domain automaton.

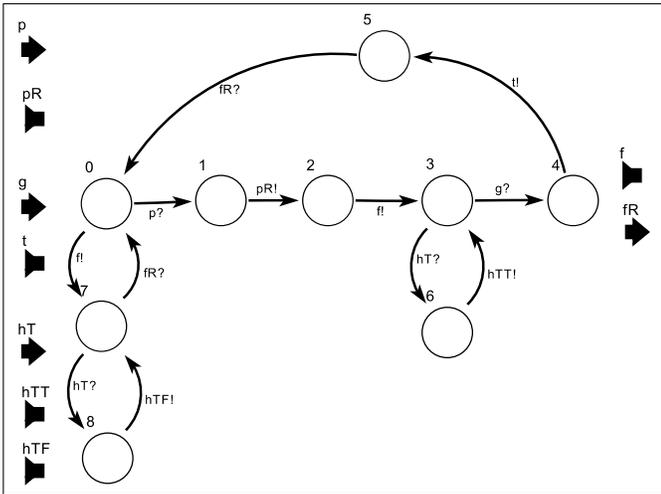


Fig. 9. DE Domain automaton.

To check the compatibility of the above two actors with the SDF and DE domains, we can compose the actor automata with the domain automata. In the theory of interface automata, two automata are compatible if their composition is not empty. The composition of the SDFactor with the SDF domain automaton is shown in Fig. 10. It is not empty, so the SDF actor can be used in the SDF domain. However, the composition of the SDFactor with the DE domain automaton is empty, so the SDF

actor cannot be used in the DE domain. This is because the actor may call `get()` when there is no token in the receiver, and this call is not accepted by an empty DE receiver.

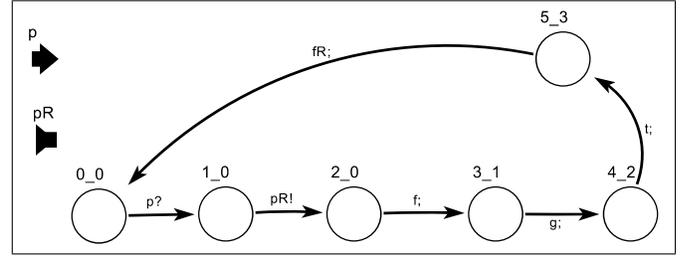


Fig. 10. Composition of the SDF Domain and the SDF actor automata.

Now let us compose the PolyActor in Fig. 7 with the SDF and DE domain automata. The results are shown in Fig. 11 and Fig. 12. Since these compositions are not empty, we have verified that PolyActor can work in both domains.

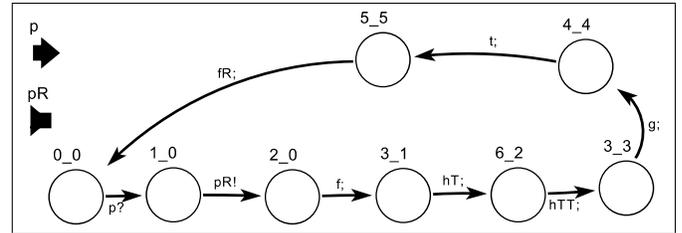


Fig. 11. Composition of the SDF Domain and the domain-polymorphic actor automata.

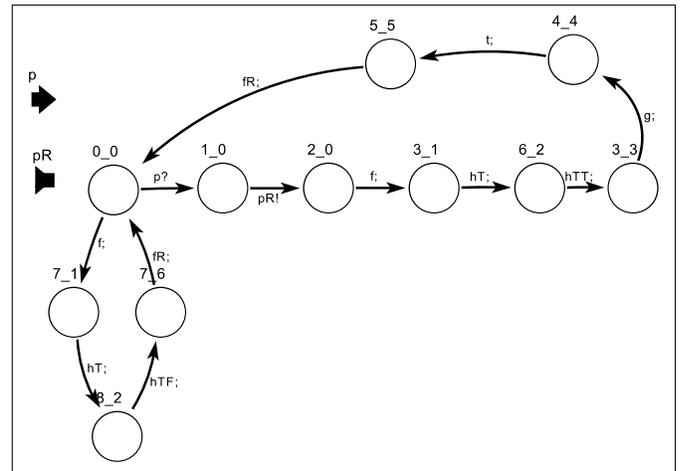


Fig. 12. Composition of the DE Domain and the domain-polymorphic actor automata.

Although the above suggests composing the automaton model of an actor with each of the domain automata for checking compatibility, there is a much more elegant way to characterize the domain-polymorphic behavior of actors. To do this, we can take advantage of the alternating simulation relation of interface automata. For example, there is an alternating simulation relation from SDFDomain to DEDomain. This is indicated in Fig. 13, which depicts this relation, as part of a partial order on domain automata. In this diagram, DE is above SDF because there is an alternating simulation relation from SDF to DE. The

other automata in this diagram are omitted in this paper for the sake of brevity.

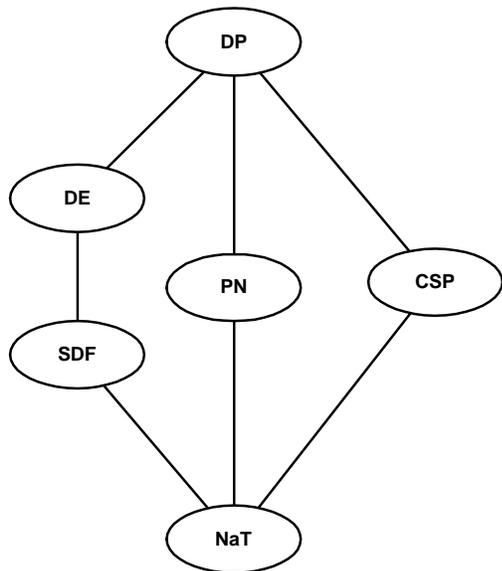


Fig. 13. An example of the partial order of domain automata.

By applying a theorem of interface automata, if an actor automaton is compatible with a certain domain automaton P , it is also compatible with all the other domain automata from which there is an alternating simulation relation to P . Therefore, once we have verified that PolyActor is compatible with $DEDomain$, we have also established that it is compatible with $SDFDomain$. So in fact, we do not need to compute the composition in Fig. 11. The top element in Fig. 13 is the Domain Polymorphic (DP) automaton, to which there is an alternating simulation relation from all the other automata. If an actor is compatible with DP, it is then compatible with all the domains below. The exact design of the DP automaton is part of our future research.

VI. GENERATING CODE

A. Translating models

Formal system models are used in several ways. At their most basic, formal models can be used to prototype and validate new designs. After the model has been created and validated, the model is typically translated by hand into code for a system implementation. This code is written in an appropriate target architecture-specific language that compiles directly onto the target architecture. If the system being designed is an application-specific integrated circuit, then the implementation code will often consist of a hardware description language, such as VHDL or Verilog. Similarly, if the system is a microprocessor system, then the implementation code will likely be some flavor of C [26]. If the system is heterogeneous, then more than one language is usually used.

However, this translation process is complex and prone to error, which reduces the value of any testing and verification done on the model. A more robust strategy is to follow the precepts of *model-based design* [44], where the model is used as the specification for the system, and the realization is synthesized from the model. This is sometimes called *code generation*. In this

case, implementation code is generated directly from the model by a *code generator*. In order to generate the most efficient code possible, this generator is usually written with a particular type of model and target architecture in mind [46], thus following the precepts of platform-based design [43]. To handle hierarchical models, the model can either be flattened, resulting in a large, generally amorphous model, or code can be recursively generated at each level of the hierarchy, incorporating the output of code generation at one point in the hierarchy into code generated at a higher level.

B. Domain Optimizations

A key part of a code generator that creates efficient code is analyzing the model to determine how it can best be transformed into implementation code. This analysis must take into account both the control flow and communication between actors. This information is readily available in a Ptolemy domain, allowing transformations to be easily written.

For example, the sizes of communication queues in an SDF domain model can be statically determined, since the actors are statically scheduled. Replacing the queues with circularly addressed arrays usually results in more efficient implementation code [8]. Furthermore, since the order that actors are fired is known during code generation, optimizations can be made between actors to improve the generated code. For a model in the PN domain, the number of executing processes can often be reduced by consolidating chains of actors together into a single process. This is often useful if the system is being implemented on a microprocessor using expensive operating system threads.

These optimizations might be possible for an amorphous model, although the analysis required would be much more complex. Using hierarchical heterogeneity, the model of computation is specified by the designer as part of the design and a design tool needs only make use of that information. In an amorphous model, as the complexity of a system model grows, and as the types of communication styles allowed are increased, the analysis necessary to determine whether a particular optimization is possible must also increase. On the other hand, when composing models of computation, the analysis and transformations during code generation do not individually increase in complexity, but are simply applied hierarchically.

VII. AN EXAMPLE FROM HIGH ENERGY PHYSICS

In order to better illustrate the utility of hierarchically heterogeneous design, we describe the model of a data acquisition (DAQ) system designed for high energy physics experiments. The goal of this system is to capture information from a particle detector, which is made up of a large number of individual sensors. The DAQ system also preprocesses the information to identify the relatively small amount of data which is experimentally significant. After filtering out the data that is not useful, the remaining data from each individual sensor is aggregated into a single stream for storage and later off-line analysis. Because of the large numbers of sensors (often several thousand), and high sample rates (10^8 samples per second), this processing

and aggregation must be done in real time. Furthermore, understanding and modeling the physics of the problem is important for making good system-level tradeoffs.

The DAQ system model presented in the following sections was primarily designed for neutrino astrophysics experiments, although the components of the system can be reused for other physics experiments. The model includes an *event generator* used to simulate properties of the physical events under observation and the geometry of the detector, a *sensor model* used to simulate the properties of individual sensors, an *analog DAQ front end model* used to simulate the analog signal processing, and sampling of the input signal, and a *digital DAQ back end model* that simulates signal quantization, digital signal processing and data losses due to buffer overflow during aggregation. We briefly summarize the physics of the experiment and show how the system can be modeled using a hierarchically heterogeneous model. A more complete description of the system can be found in [35].

A. Neutrino Astronomy Experiments

Neutrino astrophysics experiments are intended to detect high energy neutrinos in the cosmic ray flux. Neutrinos are very weakly interacting particles and can penetrate galactic dust clouds and even the entire earth without being stopped or losing energy. Despite their weak interactions, neutrinos can interact with nucleons and generate high energy muons. These charged particles can be detected more easily. At very high energies ($> 100\text{GeV}$) the muon's momentum is approximately half the neutrino energy. At these energies, the neutrino induced muon flux is so small that detectors have to cover areas of one km^2 with volume on the order of one km^3 or more to detect astrophysical neutrinos. One of the few known ways to instrument a large detector uses optical detection of Cherenkov radiation. Cherenkov radiation is electro-magnetic radiation generated by charged particles (in this case the muons) moving faster through a polarizing medium than the local speed of light in the medium. The power spectrum of Cherenkov radiation is proportional to the frequency of the emitted photons. The very faint Cherenkov radiation is detected by large diameter (8"-12") photo multiplier tubes (PMTs) enclosed by pressure resistant optical module (OM) spheres. A vertical string of detector modules intercepts a Cherenkov cone along a cone section, i.e. a wedge or a hyperbola. This intersection creates a characteristic time profile along the string: the relative timing of photons registered by OMs is a function of the position of the OM and the direction of the particle track. Detecting this radiation, correlated across a number of sensors in the string allows extrapolation of the original path of the neutrino. Particles emit Cherenkov radiation under a fixed angle α . If the distance between the particle track and the string is called r , the projection of the nearest point onto the string coordinate (here chosen to be z) is named z_0 , the angle of the track relative to the string (the zenith angle) is called θ , and c denotes the speed of light, the equation for the hit time $t(z)$ becomes:

$$t(z) = (z - z_0) \cos(\theta) + \frac{\sqrt{r_0^2 + (z - z_0)^2 \sin^2(\theta)}}{\tan(\alpha)} \quad (1)$$

While this geometric model is an oversimplification of the physics, it is sufficient for an initial exploration for systems engineering purposes. A Ptolemy model calculating the event arrival times is shown in Figure 14.

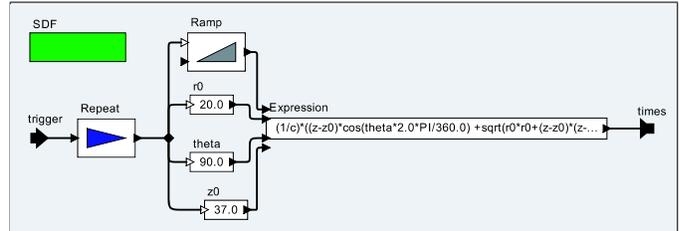


Fig. 14. Model that calculates arrival time of Cherenkov radiation.

Given the parameters of the neutrino path, this synchronous dataflow model creates a sequence of coordinates for eight equidistant modules along a detector string and calculates the corresponding hit times for each module. These times are used to generate events in the top level discrete event system model (shown in Figure 15). These events simulate the arrival of Cherenkov radiation at the photomultiplier tubes at the correct times. Note that the cone2time model itself executes in zero time and we have simply used the synchronous dataflow model as a convenient way of organizing the computation of the event times. The calculated times are converted into timed discrete events by the *TimedDelay* actors. The times computed by this model for an interesting neutrino path are plotted in Figure 16.

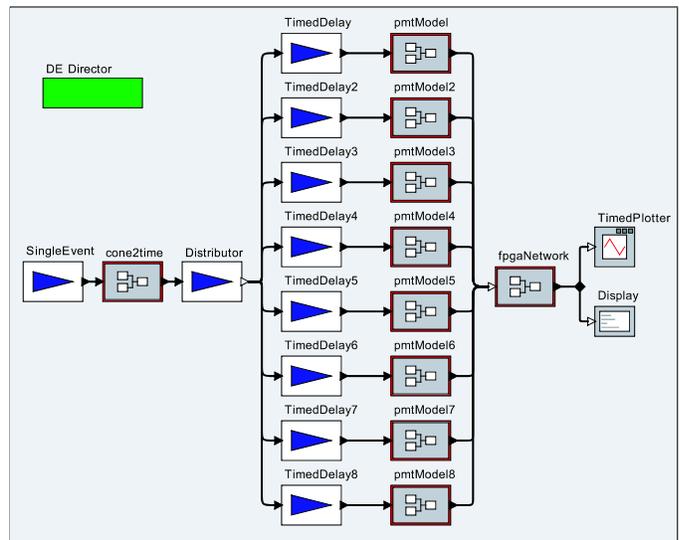


Fig. 15. Complete system model.

B. Sensor Model

Photo-multiplier tubes are stochastic amplifiers: a photon can create a single photo electron at the surface of the photocathode which is then accelerated towards an electron multiplier chain by a strong electrostatic field where it creates multiple secondary electrons. The *pulse height distribution* (PHD) is the most important performance characteristics of a PMT⁵. A

⁵A good PHD is nearly Gaussian with width $\sigma_{PHD}^2 \approx 1$. Individual photoelectrons are also delayed by a random drift time due to the inhomogeneity

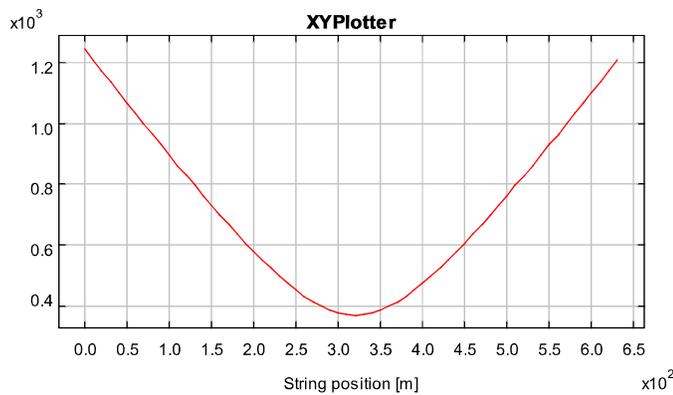


Fig. 16. The arrival time of Cherenkov radiation[ns] vs. string position for a horizontal track 100 meters from the string.

discrete-event model of the stochastic amplification properties of a PMT is shown in Figure 17. This model simply applies a random amplitude and delay to the incoming event.

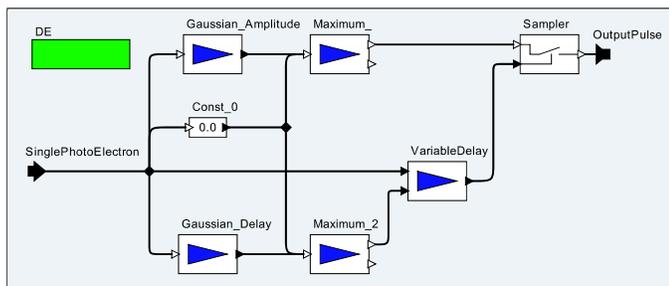


Fig. 17. Discrete event model of the stochastic amplification properties of a PMT.

C. Front end

A typical DAQ front end consists of a pre-amplifier, a signal shaper to limit the bandwidth of the circuit and improve the signal-to-noise ratio of detector signals, a sample/hold (S/H) circuit, and an analog-to-digital converter (ADC)⁶.

The continuous time PMT waveform and pre-amplifier model in Figure 18 uses a single fourth order low-pass filter with $5ns$ time constant, corresponding to $\approx 60MHz$ pre-amplifier bandwidth. The first pole is given explicitly with an integrator and feedback loop, while the remaining three are lumped into a simple Laplace transfer function. The dynamics of such a system can be specified in other ways, of course, but this representation both naturally describes the continuous time dynamics of the PMT, while emphasizing the lack of synchronization between input events and the sampling of the signal. The pre-amplified

of the accelerating electrostatic field. The average drift time ($\approx 20ns$) and the time spread ($\approx 2ns$) of the drift time distribution can be modeled with a normal distribution $N(t_{drift}, \sigma_{drift}^2)$.

⁶Most of the information about the path of an incoming neutrino is carried by the arrival time of Cherenkov radiation at each PMT. The arrival time can be extracted from a PMT pulse by sampling it with high resolution (12 bit) at a high rate (100MSPS) and fitting a parameterized analytical expression for the expected waveform to the sampled data. For PMT pulses three free parameters (amplitude, offset and a pulse time) are used and three samples are sufficient to calculate the pulse parameters. A PMT pulse with 15-20ns width gives 3-5 non-zero samples at 100MSPS sampling rate. The statistical error for the time

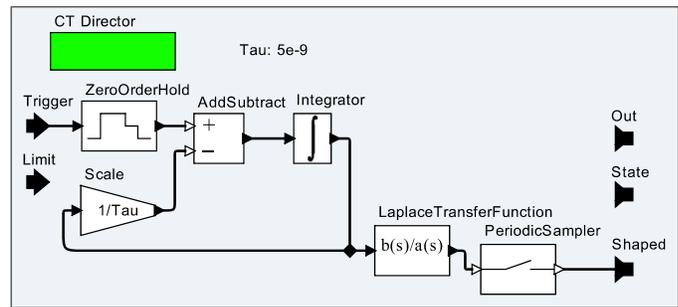


Fig. 18. Continuous time model of the PMT pulse shape.

and shaped waveforms are sampled at 10ns intervals and the samples are discretized to the ADC resolution of 12 bit, i.e. an integer between 0 and 4095.

An important issue for mixed DE/CT simulation is efficiency. While a DE simulation is only evaluated at times at which at least one variable is changing, CT models are evaluated continuously at time intervals which are set by the differential equation solver of the simulator. The use of a periodic sampling actor as a S/H stage model also requires the simulator to evaluate the model every 10ns. This is very inefficient since the average hit rate of a PMT is 1kHz and the PMT pulse is only 100ns long. A free running CT model would create an unnecessary computational overhead. By embedding it into a *modal model*, a *finite state machine* (shown in Figure 19) determines when the CT model executes. This embedded modal model contains a state machine with two states, an inactive *Init* state and a *Pulse* state which activates the CT model. The transition from the *Init* to the *Pulse* state is triggered by the presence of a trigger signal, the transition back into the *Init* state occurs as soon as the output of the CT model falls below a given threshold. This allows low overhead simulation without losing relevant information.

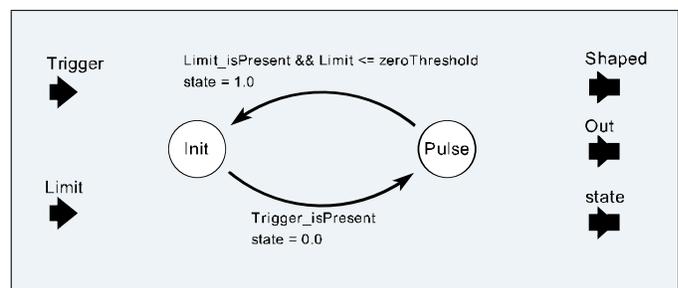


Fig. 19. Controller of the PMT and DAQ front end.

The model combining the stochastic and continuous time properties of each PMT is shown in Figure 20. This model also contains the digital signal processing aspects of the system, which are described using hierarchical synchronous dataflow models *12 bit quantization* and *trigger detection*. This model generates output events that consist of an array of four 12-bit samples, corresponding to samples at the leading edge of a PMT pulse.

parameter of the fit is usually less than 20% of the sampling period, or about 2ns.

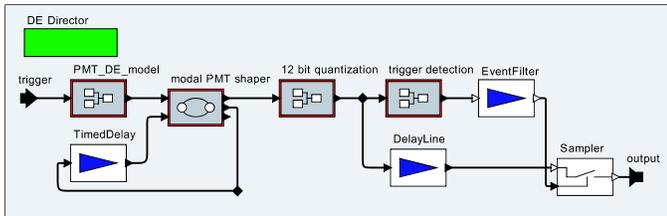


Fig. 20. Model of the PMT and DAQ front end.

D. Back End Model

The communication of data and control messages between the DAQ front end (analog electronics and ADC), the digital signal processing and the final storage (on a CPU farm) is accomplished with a static network of transmitting and receiving nodes⁷. Figure 21 illustrates the network structures of an FPGA. This network uses a custom datagram protocol which

high data rate of the router and combiner. All of the network circuits are modeled using synchronous dataflow models with finite state machines to track the states of the packet protocol.

The discrete-event model of the network of FPGAs is shown in Figure 22. This model shows the transmitters for each of the eight DAQ channels, along with the combiners that merge the incoming packet streams. A discrete-event model is used instead of a more synchronous model to allow explicit modeling of the clock using a finite state machine. This improves the simulation speed of the DE/SDF combination when the network is known to be idle.

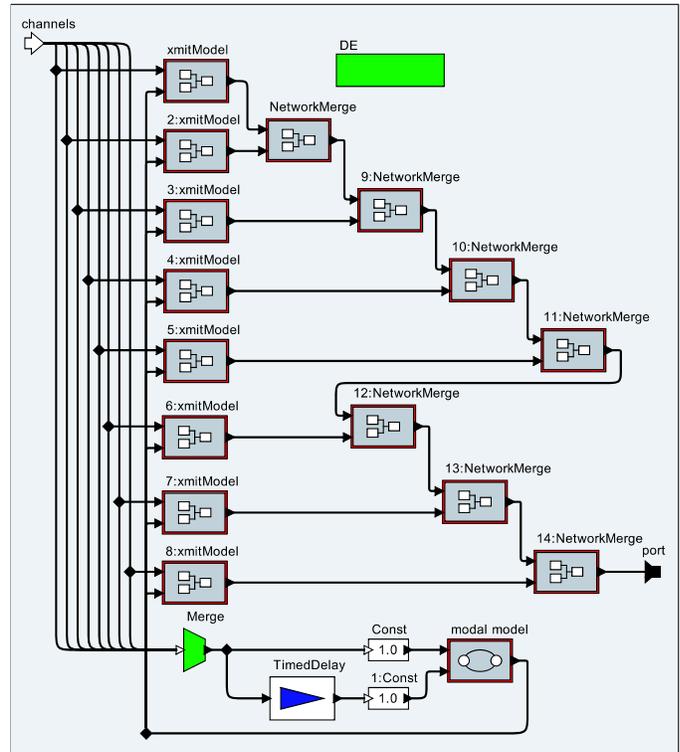


Fig. 22. Model of a network of FPGAs.

can be implemented in a small fraction of available FPGA logic. A simple, high performance packet routing strategy with static routing tables is used to transport messages between nodes of the hardware network. *Router circuits* split nets into subnets. A router reads every packet destination address and reroutes the packet to the internal network if the packet's destination address matches the address of the local subnet. Data from multiple subnets are merged by *combiner circuits* into one output stream. Since the transmitter architecture is non-blocking, a combiner uses FIFOs on its inputs to buffer incoming data until the output line becomes available. The size of the FIFOs are statically determined, which may cause packets to be dropped. The *transmitter circuits* and *receiver circuits* present a parallel interface to the network which is asynchronous from the

This heterogeneous model allows a designer to investigate two particular aspects of the design: the design of the trigger detection circuit used in Figure 20, and the network topology which is exemplified by Figure 22. Optimal design of both of these aspects requires in-depth understanding of the innate heterogeneous nature of the system, from underlying physics and continuous dynamics of the detector to digital signal processing and network design. We believe that hierarchically heterogeneous models, such as this one, can greatly help understanding the interactions between heterogeneous aspects of the system, ultimately leading to better designs. Furthermore, the above model efficiently and accurately simulates the operation of a real system across time granularity differences of nearly six orders of magnitude.

VIII. CONCLUSION

In this paper, we have proposed an actor-oriented, hierarchically heterogeneous approach as a way to minimize emergent behavior in complex models, such as those of modern embedded systems. This makes designs easier to understand, and the

⁷The digitized signals from eight analog channels are fed into one FPGA which writes them into 27 Mbit DDR SRAM waveform memories. Digital signal processing algorithms inside the FPGA are used to extract trigger information such as event energy and timing. Processing of the *GByte/s* data stream is continuous and dead-time-free. Eight FPGAs are connected in a ring topology which is favorable due to the simplicity of the board design and the nearly optimal electrical line length, which helps minimizing noise problems in the analog section.

additional structure imposed by the hierarchy may be used for other purposes as well, such as analysis and code generation.

The central notion in our concept of hierarchical model decomposition is that of a *domain*, which implements a particular model of computation. Technically, a domain serves to separate the flow of control and data between components from the actual functionality of individual components. Besides facilitating hierarchical models, this factoring potentially also dramatically increases the reusability of components and models.

Key issues to this approach are the compatibility of actors with domains, and the compositionality of domains. To characterize the dynamic interaction between actors and domains, we use interface automata to model the behaviors of actors, receivers, and directors. The compositionality of these automata precisely defines the compatibility of actors with domains. The notion of *domain polymorphism* characterizes the reusability of actors.

Starting from the work presented here, future research is needed on many aspects. For example,

- The diversity of embedded system applications gives rise to a wide range of domains. Building frameworks that help deep understanding of these domains and their implications on analysis and code generation will be a part of the Ptolemy project.
- The interface automaton approach for characterizing interactions among actors and domains is a starting point for fully developing a behavioral type system. We have also proposed using automata to do on-line reflection of component states. In addition to run-time type checking, the resulting reflection automata can add value in a number of ways. For example, in a reconfigurable architecture or distributed system, the state of the reflection automata can provide information on when it is safe to perform mutation. Reflection automata can also be valuable debugging tools.
- Migrating high-level models to embedded system implementations is essential for truly applying the hierarchical heterogeneous design methodology to real world applications. Besides the code generation facilities discussed in section VI, how to characterize embedded system architectures, and how to abstract an application platform to better support hierarchical run-time systems are in the critical path.

ACKNOWLEDGMENTS

This research is part of 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.

REFERENCES

- [1] M. Adellantado and F. Boniol. Controlling real-time asynchronous tasks with Esterel synchronous language. *Principles of Distributed Computing*, page 387, 1994.
- [2] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, 1986.
- [3] P. Alexander, C. Kong, and D. Barton. Rosetta semantics strawman, September 2000. Available at <http://www.sddl.org>.
- [4] P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry. *Hybrid Systems II*. Lecture Notes in Computer Science 999. Springer-Verlag, 1995.
- [5] Darrell Barker. Requirements modeling technology: A vision for better, faster, and cheaper systems. In *Proceedings of the VHDL International Users Forum Fall Workshop (VIUF'00)*, Orlando, Florida, Oct. 2000.
- [6] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and realtime systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.
- [7] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.
- [8] Shuvra S. Bhattacharyya, Pravin K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.
- [9] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.
- [10] Lee Breslau et al. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.
- [11] Ed Brinksma and Tommaso Bolognesi. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.
- [12] Joseph Buck and Radhu 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.
- [13] Joseph 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.
- [14] J. Davis et al. Ptolemy II - heterogeneous concurrent modeling and design in Java. Memo M99/40, UCB/ERL, EECS UC Berkeley, CA 94720, July 1999.
- [15] Luca de Alfaro and Thomas A. Henzinger. *Interface Automata*. 2001.
- [16] A. Doholi and R. Vemuri. The definition of a VHDL-AMS subset for behavioral synthesis of analog systems. In *Proc. of IEEE/VIUF BMAS*, 1998.
- [17] Robert Esser and Jörn W. Janneck. Moses: A tool suite for visual modeling of discrete-event systems. In *Symposia on Human-Centric Computing (HCC '01)*, pages 272–279. IEEE Computer Society, September 2001.
- [18] T. Grotker, R. Schoenen, and H. Meyr. Pcc: A modeling technique for mixed control/data flow systems. In *Proc. of the European Design and Test Conference (ED&TC)*, 1997.
- [19] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal: A data flow oriented language for signal processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 34:362–374, 1986.
- [20] 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.
- [21] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [22] C. A. R. Hoare. A theory of CSP. *Communications of the ACM*, 21(8), August 1978.
- [23] C. A. R. Hoare. *Communicating sequential processes*. Computer Science. Prentice Hall International, 1985.
- [24] Gilles 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.
- [25] Gilles Kahn and David 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.
- [26] Brian W. Kernigan and Dennis M. Ritchie. *The C Programming Language, 2nd Ed.* Prentice-Hall, 1988.
- [27] J. Kunkel. COSSAP: A stream driven simulator. In *IEEE Int. Workshop Microelectron. Commun., Interlaken, Switzerland*, March 1991.
- [28] Rudy Lauwereins, Marc Engels, Marleen Ad, and J. A. Peperstraete. Grape-II: A system-level prototyping environment for dsp applications. *IEEE Computer*, 28(2):35–43, 1995.
- [29] Bilung Lee. *Specification and Design of Reactive Systems*. PhD thesis, EECS Department, University of California at Berkeley, CA, 2000.
- [30] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, pages 55–64, September 1987.
- [31] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–798, May 1995.
- [32] Edward A. Lee and Yuhong Xiong. System-level types for component-based design. In T.A. Henzinger and C.M. Kirsch, editors, *Proceedings*

- of *EMSOFT 01: Embedded Software*, Lecture Notes in Computer Science 2211, pages 148–165. Springer-Verlag, 2001.
- [33] J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S. Sastry, and E. A. Lee. A hierarchical hybrid system model and its simulation. In *38th IEEE conference on Decision and Control, Phoenix, AZ*, December 1999.
 - [34] Jie Liu. Continuous time and mixed-signal simulation in Ptolemy II. Memo M98/74, UCB/ERL, EECS UC Berkeley, CA 94720, July 1998.
 - [35] Jozsef Ludvig, James McCarthy, Stephen Neuendorffer, and Sonia R. Sachs. Reprogrammable platforms for high-speed data acquisition. *Journal of Design Automation for Embedded Systems*, 2002. To appear.
 - [36] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, October 1991.
 - [37] R. Milner. A calculus for communicating systems. Lecture Notes in Computer Science 92. Springer-Verlag, 1980.
 - [38] Mohand Mokhtari and Michel Marie. *Engineering Applications of MATLAB 5.3 and SIMULINK 3*. Springer Verlag, 2000.
 - [39] Pieter J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *Hybrid Systems: Computation and Control (HSCC99)*, Lecture Notes in Computer Science 1569, pages 148–165. Springer-Verlag, 1999.
 - [40] W. Nagel. Spice 2—a computer program to simulate semiconductor circuits. Memo M520, UCB/ERL, EECS UC Berkeley, CA 94720, 1975.
 - [41] Z. Navabi. *VHDL Analysis and Modeling of Digital Systems*. McGraw Hill, Inc., 1993.
 - [42] M. Richard and O. Roux. An attempt to confront asynchronous reality to synchronous modelization in the Esterel language. *Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 571*, pages 429–450, 1992.
 - [43] Alberto Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign*, February 2002.
 - [44] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *IEEE Computer*, pages 110–112, April 1997.
 - [45] D. E. Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, USA, 1991.
 - [46] Pieter van der Wolf, Paul Lieverse, Mudit Goel, David La Hei, and Kees Vissers. An MPEG-2 decoder case study as a driver for a system level design methodology. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*. SIGDA, ACM, May 1999.
 - [47] John Wexler. *Concurrent Programming in Occam 2*. Ellis Horwood Series in Computers and Their Applications, England, 1989.