Addressing Modeling Challenges in Cyber-Physical Systems

Patricia Derler
Edward A. Lee
Alberto L. Sangiovanni-Vincentelli

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-17
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-17.html

March 4, 2011

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0931843 (ActionWebs), and #1035672 (CSR-CPS Ptides)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.}

Addressing Modeling Challenges in Cyber-Physical Systems

Patricia Derler, Edward A. Lee, and Alberto Sangiovanni Vincentelli

March 4, 2011

Abstract

This paper focuses on the challenges of modeling cyber-physical systems that arise from the intrinsic heterogeneity, concurrency, and sensitivity to timing of such systems. It uses a portion of an aircraft vehicle management systems (VMS), specifically the fuel management subsystem, to illustrate the challenges, and then discusses technologies that at least partially address the challenges. Specific technologies described include hybrid system modeling and simulation, concurrent and heterogeneous models of computation, the use of domain-specific ontologies to enhance modularity, and the joint modeling of functionality and implementation architectures.[1]

1 Introduction

Cyber-physical systems (CPS) are integrations of computation and physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. The design of such systems, therefore, requires understanding the joint dynamics of computers, software, networks, and physical processes. It is this study of *joint* dynamics that sets this discipline apart.

When studying CPS, certain key problems emerge that are rare in so-called general-purpose computing. For example, in general-purpose software, the time it takes to perform a task is an issue of *performance*, not *correctness*. It is not incorrect to take longer to perform a task. It is merely less convenient and therefore less valuable. In CPS, the time it takes to perform a task may be critical to correct functioning of the system.

In CPS, moreover, many things happen at once. Physical processes are compositions of many things occuring at the same time, unlike software processes, which are deeply rooted in sequential steps. Abelson and Sussman [1] describe computer science as "procedural epistemology," knowledge through procedure. In the physical world, by contrast, processes are rarely procedural.

[1]The authors are with the EECS Department of UC Berkeley.
Contact them at {pd, eal, alberto}@eecs.berkeley.edu.

As an experiment, the models discussed in this paper are available in executable form using Web Start at http://ptolemy.org/modelingCPS. For convenience, the figures in the paper that show models include hyperlinks directly to the model. While viewing the PDF file, just click on the figure.

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0931843 (ActionWebs), and #1035672 (CSR-CPS Ptides)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

Physical processes are compositions of many parallel processes. Measuring and controlling the dynamics of these processes by orchestrating actions that influence the processes are the main tasks of embedded systems. Consequently, concurrency is intrinsic in CPS. Many of the technical challenges in designing and analyzing embedded software stem from the need to bridge an inherently sequential semantics with an intrinsically concurrent physical world.

The major theme of this paper is on *models* and their relationship to realizations of cyber-physical systems. The models we study are primarily about dynamics, the evolution of a system state in time. We do not address structural models, which represent static information about the construction of a system, although these too are important to system design.

Working with models has a major advantage. Models can have formal properties. We can say definitive things about models. For example, we can assert that a model is deterministic, meaning that given the same inputs it will always produce the same outputs. No such absolute assertion is possible with any physical realization of a system. If our model is a good abstraction of the physical system (here, "good abstraction" means that it omits only inessential details), then the definitive assertion about the model gives us confidence in the physical realization of the system. Such confidence is hugely valuable, particularly for embedded systems where malfunctions can threaten human lives. Studying models of systems gives us insight into how those systems will behave in the physical world.

In this paper, we begin in Section 2 by considering a vehicle management system (VMS) challenge problem provided to us by the United Technologies Corporation as support to our MuSyC research effort (Multiscale System Center). We focus on the fuel management subsystem, using it to illustrate modeling challenges that arise from the intrinsic heterogeneity, concurrency, and sensitivity to timing of such systems. In Section 3, we discuss a few techniques that at least partially address these challenges, and we identify open research challenges. In Section 4, we analyze the state of the art in existing tools and methods.

2 Modeling Challenges

In model-based design [72] and model-driven development [71], models play an essential role in the design process. They form the specifications for systems and reflect the evolution of the system design. They enable simulation and analysis, both of which can result in earlier identification of design defects than prototyping. Automated or semi-automated processes can, under certain circumstances, synthesize implementations from models. But the intrinsic heterogeneity and complexity of CPS stresses all existing modeling languages and frameworks.

A model of a CPS comprises models of physical processes as well as models of the software, computation platforms and networks. The feedback loop between physical processes and computations encompasses sensors, actuators, physical dynamics, computation, software scheduling, and networks with contention and communication delays. Modeling such systems with reasonable fidelity is challenging, requiring inclusion of control engineering, software engineering, sensor networks, etc. Additionally, the models typically involve a large number of heterogeneous components. Composition semantics becomes central.

We will illustrate modeling challenges in CPS using a portion of an aircraft vehicle management systems (VMS), specifically the fuel management subsystem. We give small illustrative models constructed in Ptolemy II [19], a modeling and simulation environment for heterogeneous systems. Ptolemy uses an actor-oriented design approach [44] to model components that communicate via ports. Actors can execute and communicate with other actors. The rules of interaction between actors are defined by the model of computation (MoC).

In Ptolemy, an MoC is indicated and implemented by a *director*, which is a component in a model or submodel. A multiplicity of directors realizing distinct MoCs can be combined in a single hierarchical model. A region of a model governed by a single director is called a *domain*. The open-source Ptolemy software distribution includes directors that implement a variety of both mature and experimental MoCs, including Discrete Events (DE), Continuous Time (CT), Finite State Machines (FSM), Synchronous Reactive (SR), Process Networks (PN), and several varieties of dataflow. Hierarchical compositions that combine CT models with discrete domains such as FSM or DE can be used to model hybrid systems.

Modern aircraft are typically equipped with several fuel tanks and a variety of valves, pumps, probes, sensors and switches. The primary purpose of the fuel system is to reliably supply engines with fuel. Secondary functions of the fuel system include providing an engine coolant and distributing weight on an aircraft to maintain balance and optimal performance. Fuel is transferred from collector tanks to engine feed tanks, and, to maintain the center of gravity of the vehicle, fuel is transferred between storage tanks. Fuel might also be used as a heat sink for heat generated during the flight. For example, fuel is used to cool the engine oil. Raising the temperature of the fuel can also make engines operate more efficiently. Fuel venting systems are required to adjust ullage space in tanks during aircraft climbs and descends. Without such venting systems, high forces on the tanks endanger the structural integrity. Fuel pressurization is required to perform efficient refueling and to transfer fuel between tanks. The system is far from simple.

Several physical processes affect the fuel control system. Weight and density of fuel changes with the temperature of the fuel, and different fuel types have different characteristics. Measuring the fuel level is complex because one has to consider the shape of the fuel tank, the motion of the aircraft, the temperature, and the pressure. In order to measure fuel level during flight operations, a large number of sensors, including float level, diodes, capacitance sensors, and ultrasonic sensors, are used. The fuel control system monitors the values retrieved from sensor tanks, computes and displays the fuel levels and operates pumps and valves. Figure 1 shows a schematic view of a typical aircraft fuel system for a military jet aircraft.

The importance of a fuel management system that supports flight crew decisions is underscored by the experience with Air Transat Flight 236 on August 23-24, 2001 [14] (see also [75]). On that flight, a fuel leak was discovered too late and led to an emergency landing. Initially, the only indication of a problem was that the flight control system reported a low oil temperature and a high oil pressure. This was a result an increased fuel flow caused by a leak in the engine. Fuel is used as a heat sink, and the increased fuel flow cooled the engine oil more than usual. The decreased oil temperature increased its viscosity, resulting in higher oil pressure. Later, the flight control system reported a fuel imbalance between the tanks. The flight crew corrected this imbalance by transferring fuel into the leaking tank. A smarter flight control system that integrated models could have helped the flight crew detect the leak sooner, for example by using model-integrated computing [72], where the behavior of a model can be compared at run time to the measured behavior of the deployed system.

Using the fuel system as an illustration, we next discuss a few specific modeling challenges.

Challenge 1: Models with solver-dependent, nondeterminate, or Zeno behavior

A CPS may be modeled as a hybrid system where physical processes are represented as continuous-time models of dynamics and computations are described using state machines, dataflow models, synchronous/reactive models, and/or discrete-event models. Continuous-time models work with solvers that numerically approximate the solutions to differential equations. Design of such solvers is an established, but far-from-trivial art. Integrating such solvers with discrete models is a newer problem, and problems persist in many available tools.

One of the problems that can arise is that the behavior defined by a model may be nondeterminate even when the underlying system being modeled is determinate. This means that the model defines a multiplicity

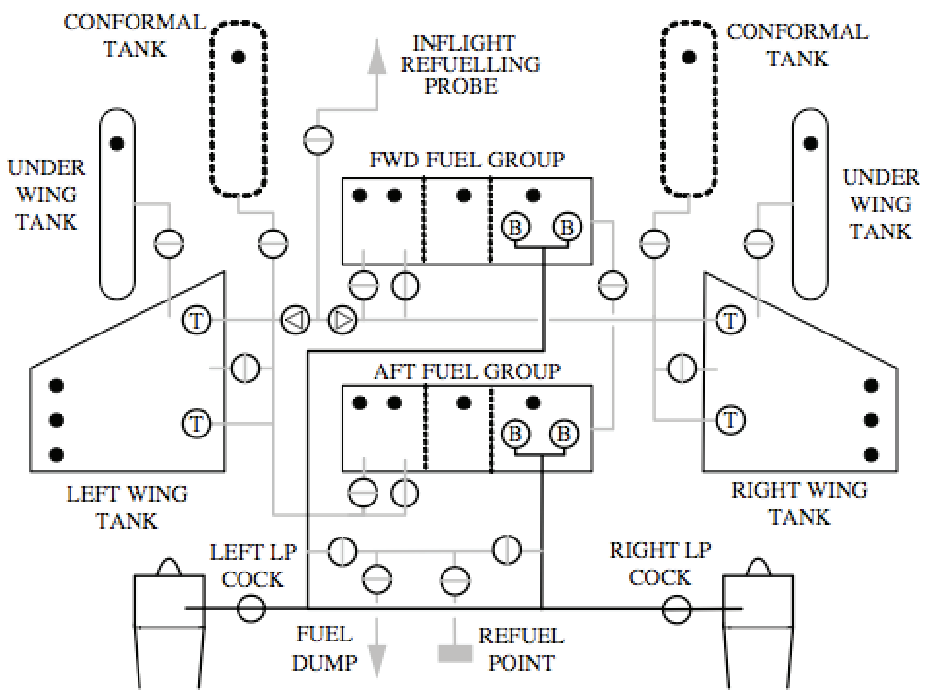


Figure 1: Military aircraft system. From [56].

of behaviors, rather than a single behavior. This can occur, for example, when discrete-events are simultaneous and the semantics of the modeling language fails to specify a single behavior. Such nondeterminism can surprise system designers. Another problem that can arise is that numerical solvers typically dynamically adjust the step size that they use to increment time, and the behavior of the model can depend on the selected step sizes. Again, this will surprise designers, who rarely have insight into the basis for step-size selection. A third problem that can arise is that models may exhibit Zeno behavior, where infinitely many events occur in a finite time interval. Such behavior from a model may reflect physical phenomena, such as chattering, but Zeno behavior can also arise as an artifact of modeling.

We can illustrate some of these difficulties using a model of a fuel tank. One challenging part of such a model is to accurately specify how the behavior changes when the tank becomes full or empty. It is reasonable to model "becoming full" and "becoming empty" as discrete events, even though the physical world is not so discrete. We do not want to force modelers to use overly detailed models of physical phenomena that are not central to their design problem, and such discrete models provide adequate abstractions.

Figure 2 depicts one possible Ptolemy II model of a single fuel tank. This model is constructed using a continuous-time director with semantics similar to popular continuous-time simulation tools such as Simulink (from The MathWorks) and the LabVIEW Control Design and Simulation Module (from National Instruments). (For example, Simulink is used in [31] to construct similar models of aircraft fuel systems.) In this MoC, components (actors) have input and output ports, in contrast to equational modeling tools such as Modelica [23], where ports are neither inputs nor outputs.

While it may seem intuitive that a fuel tank is naturally modeled with inputs and outputs — inputs provide fuel for the tank, and outputs withdraw fuel — this is actually not the case. An input to a fuel tank cannot insert fuel if the tank is full, and an output cannot withdraw fuel if the destination is full. Hence, if we want to model, for example, the transfer of fuel from one tank to another, the amount of fuel transferred per unit time must be determined collaboratively by the two tanks and a model of the intervening pumps and valves.

Figure 2 shows one possible resolution to this dilemma. Two of the input ports of the tank represent *desired input* and *desired output* flows, rather than actual flows. A third input represents the *actual output* flow. The actual output flow has to be an input to the tank model because fuel cannot flow out of a tank unless it has somewhere to go. One of the output ports represents *available output* flow, thus enabling models where the destination for the fuel refuses to accept some or all of the output flow. At the top level in Figure 2, the *available output flow* output is fed back to the *actual output flow* input, which models the situation where there is always somewhere for the fuel to go. The destination can accept any amount of fuel.

Another output port represents the *actual input* flow. If the tank is not full, then the actual input flow will match the desired input flow. But if the tank is full, then the actual input flow will differ from the desired flow.

The model in Figure 2 also shows output ports that provide the fuel level and full and empty indicators. The latter indicators are discrete events provided by LevelCrossingDetector actors.

The particular model in Figure 2 uses two Expression actors to implement what may seem to be an intuitively accurate description of how a fuel tank behaves. In this model, the tank ignores the desired input flow to the tank when the tank is full, setting the actual input flow to zero. It also stops the output flow when the tank is empty.

The behavior of the model is problematic, however. In particular, Figure 2 embeds the tank model in a simple test scenario where the desired input flow is a constant 1.0 (in some units, such as liters per second), and the desired output flow is 0.0 for three time units, followed by 2.0 subsequently. As shown in the plot in the figure, when the tank becomes empty, the model begins chattering. This behavior exposes one flaw in the model that results in solver-dependent behavior and/or the possibility of a Zeno condition. Specifically, when the tank becomes empty, actual output flow goes to zero. But an infinitesimal amount of time later, the tank is no longer empty, so the output flow can resume. But then it becomes empty again, then resumes, etc. In simulation, it appears as if time stops advancing. Only close examination of the output signals reveals the chattering behavior.

This model has several features that challenge simulation technology. First, the Expression actors have input-output relations that are not continuous. The lower one in Figure 2, for example, has the expression

```
(level > 0) ? desiredFlow : 0.0
```

The output of this actor is equal to the input if the level is greater than zero, and otherwise the output is zero. When the level hits zero, the actual behavior of this actor will depend on numerical precision and solver-chosen step sizes. Giving a precise semantics to the time at which the level hits zero is far from trivial. And controlling the behavior at the discontinuity requires collaboration between the solver and the Expression actor, something not easy to achieve in a modular way.

In addition, the LevelCrossingDetector should produce a discrete event at the precise time at which the tank becomes full or empty. This also requires careful collaboration with the solver to ensure that step sizes in time are chosen to include the time point at which the event occurs.

The chattering exhibited by the model in Figure 2 may in fact accurately represent the physical realization. One could design a fuel tank controller that closes an output valve when the tank becomes empty. Such a design could in fact chatter. A better design would limit the input and output flows when the tank becomes either full or empty. When the tank becomes full, the input flow will be limited to be no greater than the output flow, and when the tank becomes empty, the output flow will be no greater than the input flow.

Below we will give such a model, showing how it can leverage a rigorous hybrid-systems semantics to

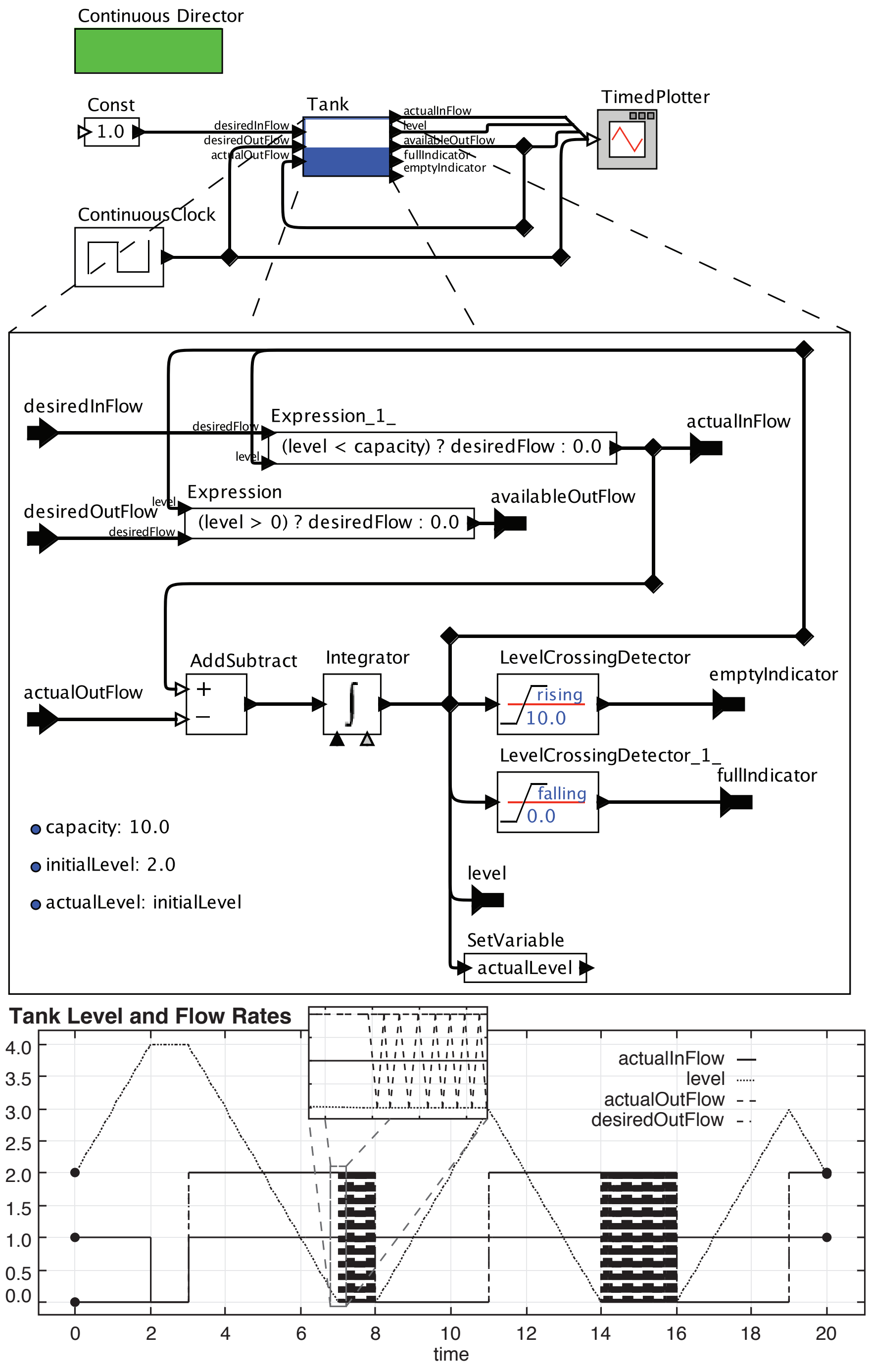


Figure 2: Naïve model of a single fuel tank in a simple test scenario.

provide a determinate and intuitive model. But before we develop the better model, we address a second challenge that concerns how to evolve and maintain a multiplicity of models and keep them consistent.

Challenge 2: Keeping model components consistent

At the previous section, we developed a simple tank model and tested it in a scenario that does not reflect any real-world scenario, but is useful nonetheless as a regression test for the tank model. In practice, unfortunately, such test scenario models are usually lost. They evolve into much more complex and realistic models, so the simple regression test disappears. If the model is kept as a regression test, then there is little assurance that the test model and the final design match. The final design is likely to be evolved from a copy of the test model, and the copy will inevitably diverge from the test model. Worse, when it comes time to design the embedded software, even if code generators are used to synthesize software from models, the software will often evolve separately from the model, increasing divergence.

The same problem arises as a simple model evolves into a complex one, where a single component in the simple model becomes multiple components in the complex one. How can we ensure that the multiple components evolve together? Until now we have only considered models with one fuel tank. Modern aircraft typically have several fuel tanks, and fuel is transferred between these tanks.

In this section, we consider the problem of evolving multiple models with multiple variants of components, all the while ensuring some measure of consistency across the models.

In the model in Figure 3, we add functionality for moving fuel between tanks. Whenever the fuel level of Tank1 falls below one fourth of its capacity, a FuelMover component transfers fuel from Tank2 to Tank1. This component uses a simple modal model to change the desired output flow of Tank2 from zero in the initial idle mode to *moveRate* (a parameter) in the moving state. The FuelMover controller specifies the desired output flow of Tank2, which in turn provides an available output flow to the desired input flow port of Tank1. Tank1 responds with an actual input flow, which in turn becomes the actual output flow of Tank2. Thus, if either tank is full or empty, the flows are appropriately limited.

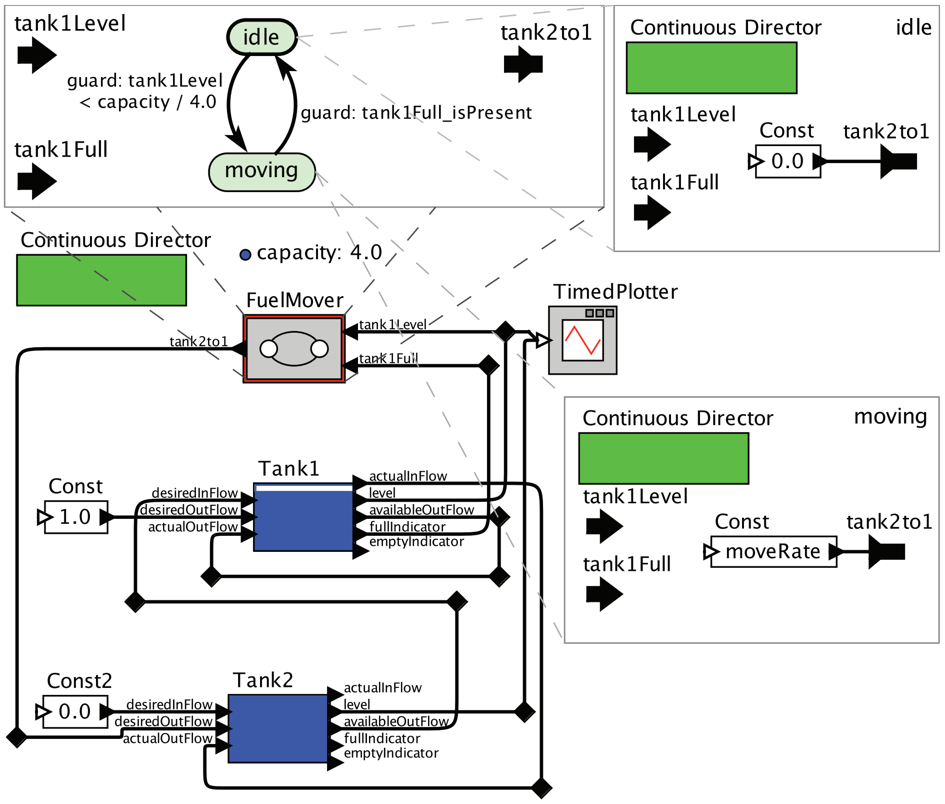
In Figure 3, the FuelMover and the two tanks are embedded in a simple test scenario where Tank1 has a constant desired output flow of 1. This could, for example, model the flow of fuel to an engine. The desired input flow of Tank1 is regulated by Tank2. Tank2 does not receive any fuel (its desired input flow is 0) and its desired output flow is regulated by the FuelMover. This component is a modal model that receives information about the fuel level of Tank1 and regulates the desired output flow of Tank2. The fuel mover can be interpreted as the cyber part of this cyber-physical system, while the Tank models the physical processes.

In a modeling environment, the tank model can be copied and re-used in various parts of the model. However, if later a change in the tank model becomes necessary, the same change has to be applied to all other tank models that were copied. This procedure is error prone, because there is no way to ensure that all copies are updated accordingly.

Below, we will explain how object-oriented design principles can be adapted to CPS models using the notion of actor-oriented classes [42], and how semantics-preserving transformation and code generation can reduce divergence in evolving designs.

Challenge 3: Preventing misconnected model components

The bigger a model becomes, the harder it is to check for correctness of connections between components. Typically model components are highly interconnected and the possibility of errors increases. We identify three types of errors: *unit errors* (1), *semantic errors* (2), and *transposition errors* (3), illustrated in Figure 4. Unit errors occur when a port provides data with different units than those expected at the receiving port.



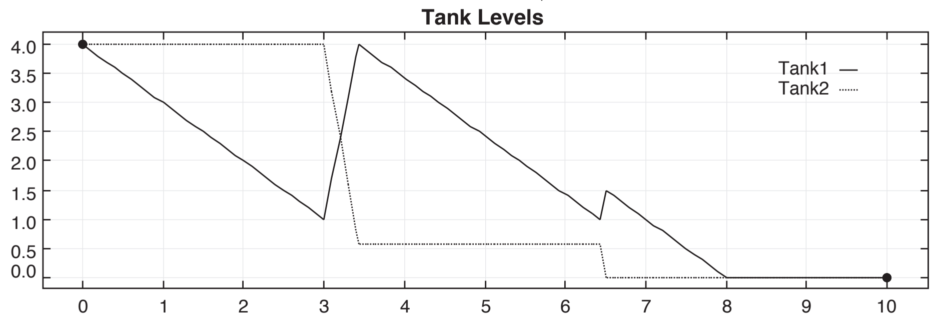


Figure 3: Fuel system with two tanks.

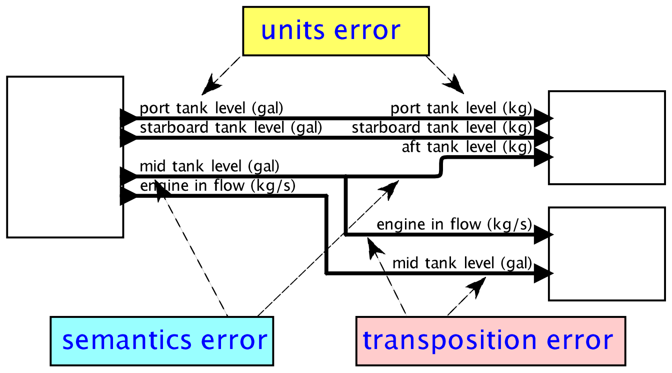


Figure 4: Misconnected model components

For example, in Figure 4, the output port of the left actor provides the fuel level in gallons, whereas the input port of the right actor expects a value in kilograms. An example of a semantic error (2) is shown, where a mid tank level is provided to a port that expects an aft tank level. A transposition error (3) occurs when connections are reversed, as in the example where the engine input flow and a tank level are reversed. None of these errors would be detected by a type system.

Below we will explain how expressing knowledge formally in an ontology [50] can encode domain knowledge and enable measures to automatically check for such errors.

Challenge 4: Modeling interactions of functionality and implementation

The models discussed above describe functionality of computations and physical processes. All of the above models ignore implementation details such as the underlying platform and the communication networks. These models implicitly assume that data is computed and transmitted in zero time, so the dynamics of the software and networks have no effect on system behavior. However, computation and communication takes time. We argue that, in order to evaluate a CPS model, it is necessary to model the dynamics of software and networks.

Implementation is largely orthogonal to functionality and should therefore not be an integral part of a model of functionality. Instead, it should be possible to conjoin a functional model with an implementation model. The latter allows for design space exploration, while the former supports the design of control strategies. The conjoined models enable evaluation of interactions across these domains.

One strategy for conjoining distinct and separately-maintained models uses the concept of aspect-oriented programming (AOP) [34]. In AOP, distinct models are maintained and developed, but they can be “woven” together to conjoin their behaviors.

AOP has been developed primarily as a software engineering technique, but the concept has also been applied to conjoining models of functionality and implementation. Specifically, below we explain how to use the concept of quantity managers, introduced in Metropolis [3], to associate functional models with implementation models.

Challenge 5: Modeling distributed behaviors

The distributed nature of CPS requires strategies that facilitate composition of components that are separated in space. Modeling distributed systems adds to the complexity of modeling CPS by introducing issues such as disparities in measurements of time, network delays, imperfect communication, consistency of views of system state, and distributed consensus. Below, we will describe a distributed programming model called PTIDES [77] that addresses many of these issues.

3 Addressing the Challenges

Over the last 20 years, we at Berkeley have been developing model-based design techniques that we believe have sufficiently well-defined semantics to provide an effective basis for platform-based design and engineering. Components can be designed to operate with a model, and when deployed, will operate in predictable ways with the deployed system. The rigorous foundations of the models [45] provide a solid basis for integration across design domains, design adaptation and evolution, and analysis and verification. Our work has been demonstrated in the open-source software frameworks Ptolemy Classic [9], Ptolemy II [19], Polis [2], Metropolis [3] and MetroII [15]. Many of the techniques that we developed have been deployed in a wide range of domain-specific applications, including hardware and FPGA synthesis, signal processing, automotive system design, computer architecture design and evaluation, instrumentation, wireless system design, mixed signal circuit design, network simulation and design, building system simulation and design, financial engineering, and scientific workflows.

In the remainder of this section, we review some of the key technologies that have been developed.

3.1 Models of Computation

Several of the challenges arise fundamentally from the complexity and heterogeneity of CPS applications. Complexity of models is mitigated by using more specialized, domain-specific models and by using modeling languages with clear, well-defined semantics. Heterogeneity necessitates combining a multiplicity of models.

To address the first of these issues, at Berkeley, we have established that models should be built using well-defined models of computation (MoCs) [16]. An MoC gives semantics to concurrency in the model, defining for example whether components in the model execute simultaneously, whether they share a notion of time, and whether and how they share state. An MoC also defines the communication semantics, specifying for example whether data is exchanged for example using publish-and-subscribe protocols, synchronous or asynchronous message transmission, or time-stamped events. We have provided a formal framework within which the semantics of a variety of models of computation can be understood and compared, and within which heterogeneous interactions across models of computation can be defined [45]. This formal foundation has been elaborated and applied to multi-clock (latency insensitive) systems [11], globally asynchronous, locally synchronous (GALS) designs [4], and to timed models of computation capable of reflecting real-time dynamics [53]. The challenge is to define MoCs that that are sufficiently expressive and have strong formal properties that enable systematic validation of designs and correct-by-construction synthesis of implementations.

3.2 Abstract Semantics

In many situations, using a single general MoC for an entire design requires giving up any possibility of property checking except by extensive simulation. More restricted (less expressive) MoCs yield better to analysis, enabling systematic exploration of properties of the design, often including formal verification.

But less expressive MoCs cannot capture the richness and diversity of complex designs. The solution is heterogeneous mixtures of MoCs. Indeed, the heterogeneous nature of most CPS applications makes multiple MoCs a necessity.

In addition, during the design process, the abstraction level, detail, and specificity in different parts of the design vary. The skill sets and design styles that engineers use on the project are likely to differ. The net result is that, during the design process, many different specification and modeling techniques will be used. The challenge is how to combine heterogeneous MoCs and determine what the composition's behavior is. Unfortunately, the semantics of different MoCs are typically not directly compatible.

A way to solve this problem is to embed the detailed models into a framework that can understand the models being composed. A theoretical approach to this view, which is well beyond the scope of this article, can be found in [10], where an abstract algebra defines the interactions among incompatible models. In some sense, we are looking at an abstraction of the MoC concept that can be refined into any of the MoCs of interest. We call this abstraction an *abstract semantics*, first introduced in [44, 37].

The inspiration on how to define the abstract semantics comes from the consideration that MoCs are built by combining three largely orthogonal aspects: sequential behavior, concurrency, and communication. Similar to the way that a MoC abstracts a class of behavior, abstract semantics abstract the semantics The concept is called a "semantics meta-model" in [69], but since the term "meta-model" is more widely used in software engineering to refer instead to models of the structure of models (see [58] and http://www.omg.org/mof/), we prefer to use the term "abstract semantics" here. The concept of abstract semantics is leveraged in Ptolemy II [19], Metropolis [3], and Ptolemy Classic [9] to achieve heterogeneous mixtures of MoCs with well-defined interactions.

3.3 Actor-Oriented Models

Model-integrated development for embedded systems [33, 30] commonly uses actor-oriented software component models [36, 44]. In such models, software components (called actors) execute concurrently and communicate by sending messages via interconnected ports. Examples that support such designs include Simulink, from MathWorks, LabVIEW, from National Instruments, SystemC, component and activity diagrams in SysML and UML 2 [5, 59], and a number of research tools such as ModHel'X [26], TDL [65], HetSC [28], ForSyDe [66], Metropolis [24], and Ptolemy II [19]. The fuel tank models given above are all actor-oriented models.

The key challenge is provide well-defined actor-oriented MoCs with well-defined semantics. All too often the semantics emerge accidentally from the software implementation rather than being built-in from the start. One of the key challenges is to integrate actor-oriented models with practical and realistic notions of time. To address Challenge 5, for example, modeling distributed behaviors, it is essential to provide multiform models of time. Modeling frameworks that include a semantic notion of time, such as Simulink and Modelica, assume that time is homogeneous in the sense that it advances uniformly across the entire system. In practical distributed systems, even those as small as systems-on-chip, however, no such homogeneous notion of time is measurable or observable. In a distributed system, even when using network time synchronization protocols (such as IEEE 1588 [29]), local notions of time will differ, and failing to model such differences could introduce artifacts in the design.

One interesting project that directly confronts the multiform nature of time in distributed systems is the PTIDES project [18]. PTIDES is a programming model for distributed systems that relies on time synchronization, but recognizes imperfections. Simulations of PTIDES systems can simultaneously have many time lines, with events that are logically or physically placed on these time lines. Despite the multiplicity of time lines, there is a well-defined and deterministic semantics for the interactions between events.

3.4 Hybrid Systems

Cyber-Physical Systems (CPS) integrate computation, networking, and physical dynamics. As a consequence, modeling techniques that address only the concerns of software are inadequate [38, 40]. Integrations of continuous physical dynamics expressed with ordinary differential equations with the discrete behaviors expressed using finite automata are known as *hybrid systems* [55]. There are many challenges in defining good hybrid systems modeling languages, a few of which are described in Challenge 1 above. These challenges include ensuring that models of deterministic systems are indeed determinate, not having behavior that, for example, depends on arbitrary decisions made by a numerical differential equation solver. It is also challenging to accurately represent in models distinct events that are causally related but occur at the same time.

At Berkeley, we have previously done a detailed study and comparison of tools supporting hybrid systems modeling and simulation [12]. Moreover, we have developed a rigorous MoC that provides determinate semantics to such hybrid systems [51, 54, 48]. This work has influenced development of commercial tools such a Simulink and LabVIEW and has been realized in the open-source tool HyVisual [7]. Moreover, we have leveraged the notation of abstract semantics to integrate such hybrid systems with other MoCs such as synchronous/reactive and discrete-event models [49]. This integration enables heterogeneous models that capture the interactions of software and networks with continuous physical processes.

We can use the rigorous hybrid systems MoC implemented in Ptolemy II to improve the tank model described in Challenge 1 above. Such a model is shown in Figure 5, which uses a modal model [46] to describe the different modes of operation based on the state of the tank. In this model, the net flow is equal to the difference between the actual input flow and the actual output flow. The model has three modes. In the normal mode, the tank is neither full nor empty, and the actual input flow equals the desired flow, and the available output flow equals the desired output flow. In the full and empty modes, either the actual input flow (if the tank is full) or the available output flow (if the tank is empty) is limited to match the other. The transitions between modes are triggered by the guards, which are predicates shown adjacent to the transitions in the figure. This model does not chatter and is determinate.

The hybrid systems semantics of Ptolemy II handles rigorously the discontinuities of the mode transitions and the event detection of the LevelCrossingDetector actors. The semantics ensures determinism in the model, helps avoid solver-dependent behavior, and helps to identify Zeno conditions.

At the bottom of 5, the plot shows the behavior of the system in a test scenario where the desired input flow is a constant 1.0, and the desired output flow alternates between zero (for 3 time units) and 2.0 (for 5 time units). Such a scenario provides a reasonable regression test for behavior of the tank model that includes full and empty conditions.

3.5 Heterogeneity

Integrating multiple MoCs such that they can interoperate, which is far from trivial, has been called "multimodeling" [57, 22, 8]. Many previous efforts have focused on tool integration, where tools from multiple vendors are made to interoperate [52, 25, 32]. This approach is challenging, however, and yields fragile tool chains. Many tools do not have adequate published extension points, and maintaining such integration requires considerable effort. A better approach is to focus on the semantics of interoperation, rather than the software problems of tool integration. Good software architectures for interoperation will emerge only from a good understanding of the semantics of interoperation.

At Berkeley, our approach has been to focus on the interfaces between MoCs. We have built a variety of modeling, analysis, and simulation tools based on different MoCs [16], and have shown how such interfaces

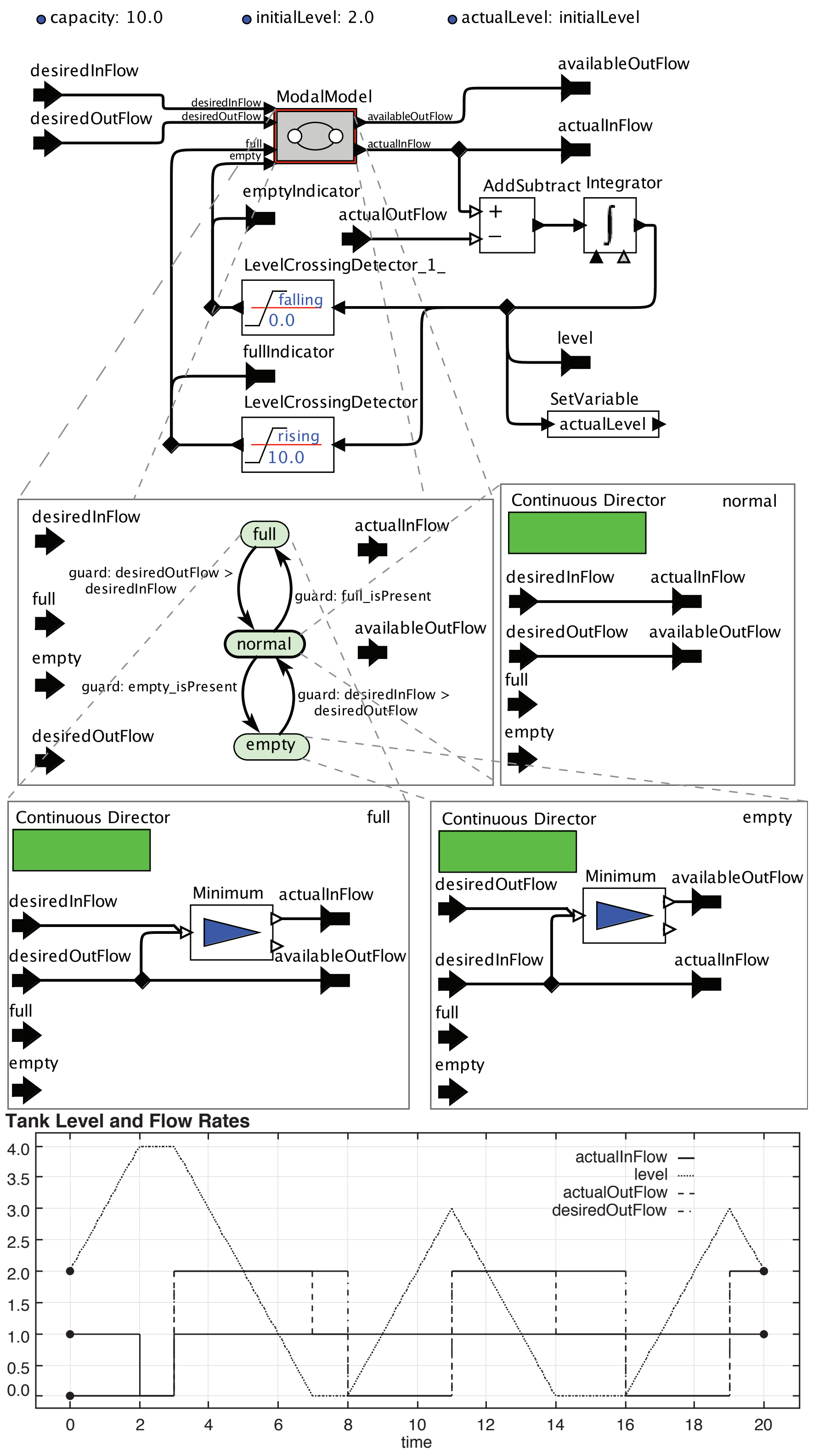


Figure 5: ModalTank: Improved model of a fuel tank using modal models.

can facilitate more robust interoperability. These include discrete-event [35] (useful for modeling networks, hardware architecture, and real-time systems), synchronous-reactive [17] (useful for modeling and designing safety-critical concurrent software), dataflow [43] (useful for signal processing), process networks [62] (useful for asynchronous distributed systems), and continuous-time models [49] (useful for physical dynamics).

For nearly all of these MoCs, the emphasis in our design has been on providing determinate behavior (where the same inputs always result in the same outputs), and introducing nondeterminacy only where it is needed by the application (for example to model faults). The result is a family of far better concurrency models than the widely used thread-based models that dominate software engineering [39].

The hybrid systems MoC of Ptolemy II is, in fact, built using such interfaces between MoCs. The modal model in Figure 5 is constructed using a Ptolemy director that has no particular knowledge of continuous-time modeling, yet it interoperates with a director that realizes continuous-time modeling. It also interoperates with other Ptolemy directors, enabling modal models to be used anywhere in a design with a wide variety of modeling styles. This stands in stark contrast to, say, Statecharts techniques, where the state machine modeling framework is designed to specifically work with exactly one concurrent MoC, typically synchronous/reactive.

Systematic support for heterogeneity can also help with Challenge 2, keeping model components consistent, because models that interoperate are less likely to diverge than models that don't. It also helps with Challenge 4, modeling interactions of functionality and implementation, because different pieces of a system can evolve non-uniformly from abstract models to detailed models of realizations. It also helps with Challenge 5, modeling distributed behaviors, because models of networks (particularly wireless networks) benefit enormously from domain-specific modeling and simulation techniques created particularly for networks. If heterogeneity is supported, then models of networks interoperate with models of the distributed components, be they mechanical, chemical, or software components.

Influenced in part by our work, SystemC, a widely used language in hardware design, is capable of realizing multiple MoCs [64, 28], although less attention in that community has been given to interoperability.

3.6 Modularity

Key to effective design of complex systems is modular design, where modules have well-defined interfaces, and composition of modules can be checked for compatibility. A key challenge is to ensure that modules are correctly composed (Challenge 3 above). It is often easy to misinterpret data provided by one module in another module. Such misinterpretation can be catastrophic. Type systems provide rudimentary protection, but much stronger protection is possible. Another key challenge is to enable reuse and customization of modules, as for example is supported by object-oriented concepts such as classes, inheritance, and polymorphism.

Like some other modeling frameworks (Modelica, SystemC, SysML, and others), Ptolemy II supports object oriented design principles in modeling, although the approach in Ptolemy II is unique [42]. In particular, we use classes to describe interface and behavior of an actor. Inheritance is used to describe relationships between objects. Instances of actors implement the behavior specified by the actor class definition. Subclasses inherit the class definitions and can be extended with additional objects as well as override parameters of the superclass. And updates to classes propagate to instances even during interactive model development.

In the fuel system models we have shown, the Tank component is defined as a class and instantiated in different parts of a model and in different models. Figure 3 displays a model with two tanks. The class definition of the tank is maintained in a separate XML file, and Tank1 and Tank2 are instances of this class. Editing the class definition results in changes in all the instances. This strategy also facilitates maintenance

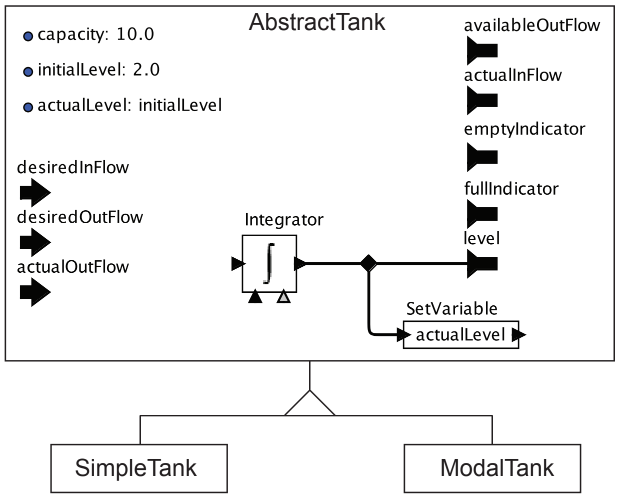


Figure 6: UML diagram for abstract and concrete tank classes.

of regression tests that can help assure that behavior remains reasonable as the tank model evolves. The regression tests are themselves models that use instances of the same class. This ensures that the model being tested and the model being developed remain consistent (Challenge 2).

Figure 6 shows a possible UML class hierarchy for the Tank models we have implemented so far. The AbstractTank class models the interface and the functionality of computing the tank level. The SimpleTank class extends the AbstractTank with the functionality shown in Figure 2. The ModalTank class represents the tank model in Figure 5. If all models, including regression tests, are developed using these class definitions, then the tank model can evolve to become more sophisticated, regression tests can check whether the evolution has changed expected behavior, and a multiplicity of instances of tank components across several models are assured to be consistent.

We have shown that object-oriented concepts such as classes, inheritance, and polymorphism can be adapted to concurrent, actor-oriented components [42]. But there are several more techniques that can applied in modeling that enhance modularity. In particular, we have also developed advanced type systems for component compositions, enabling type inference and type checking across large models with polymorphic components [76]. We have also adapted such type systems to capture domain-specific ontology information, checking for correct usage and correct interpretation of shared data [50]. And we have shown how to check for compatibility of protocols in compositions [47] and to synthesize interface adapters for separately defined components [63].

We can illustrate how a domain-specific ontology can address Challenge 3 discussed above using the techniques given in [50]. Figure 7 shows at the upper right a simple ontology for dimensions used in the fuel system. The key concepts in this ontology are Level and Flow. The goal is to prevent model builders from accidentally misinterpreting a flow as a level or vice versa.

A model can be annotated with constraints that represent the model builder's domain knowledge. Such constraints are shown at the bottom of Figure 7, where the parameters *tank2InFlow* and *tank1OutFlow* are associated with the concept *Flow*, and the parameter *capacity* is associated with the concept Level.

The system described in [50] can infer concept associations throughout a model from just a few annotations. To do this, actors may impose particular constraints. In the simple ontology shown in the figure, we have asserted that an Integrator actor converts a Flow to a Level. That is, if its output is a Level, then its input

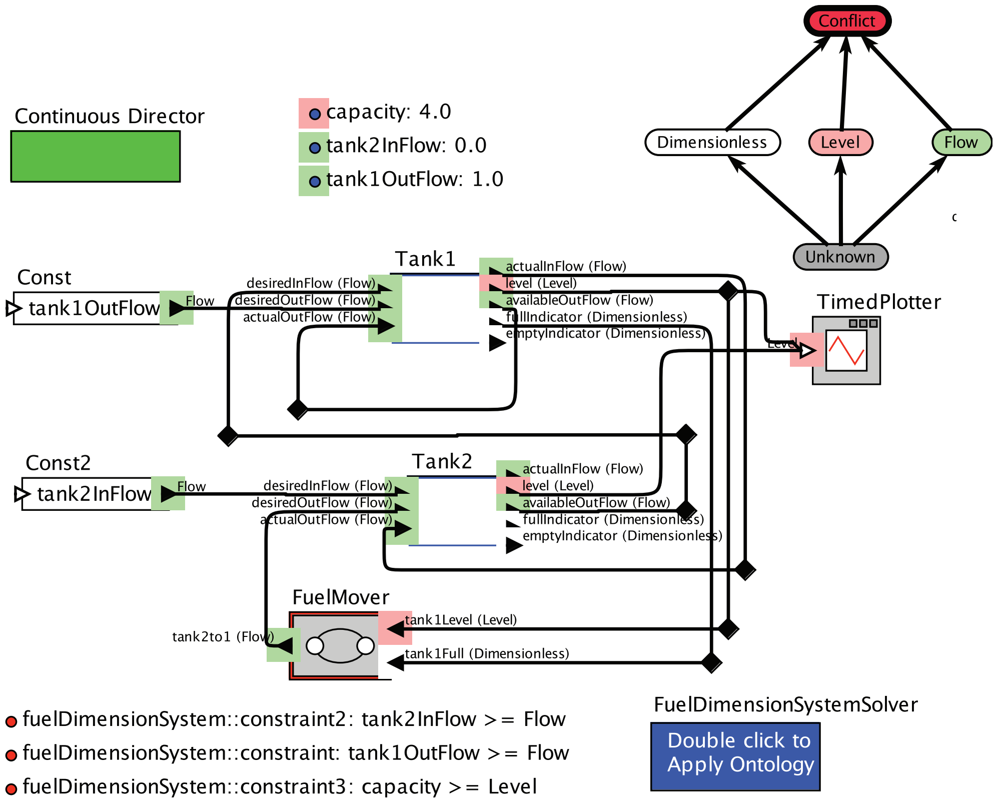


Figure 7: Domain-specific ontology and solver ensure that flows are not accidentally misinterpreted as levels and vice versa.

must be a Flow, and vice versa. Given this and similar constraints and the associations on the parameters, the ontology solver infers associations throughout the model. The result is shown as annotations close to the ports of the actors in the model, which are also color-coded according to the color scheme in the ontology. The component in the model labeled *FuelDimensionSystemSolver* handles executing the inference algorithm. With an appropriately designed ontology, errors like those in Figure 4 will be reported by the solver.

Constraints on model components such as actors impose constraints on proper relationships. Checkers such as the type system infrastructure in Ptolemy can infer these properties and detect errors. By allowing users to specify ontologies graphically and describing constraints on model elements, the domain knowledge is built up. Ontological information about the overall system can be inferred automatically from the relatively small constraint set. Inconsistencies can be detected and reported automatically.

3.7 Linking Behavior to Implementation: Quantity Managers

To support evaluation of design choices, modeling frameworks need to enable weaving together a multiplicity of models that cover different aspects of a system (Challenge 4 above). For example, a choice of networking fabric will affect temporal behavior, power usage, and vulnerability to faults. A key challenge is that this requires bridges between models that are modeling very different aspects of the same system.

The Metropolis project [3, 15] introduced the notion of "quantity manager," a component of a model that functions as a gateway to another model. For example, a purely functional model that describes only idealized behavioral properties of a flight control system could be endowed with a quantity manager that binds that functional model to a model of a distributed hardware architecture using a particular network fabric. By binding these two models, designers can evaluate how properties of the hardware implementation affect the functional behavior of the system. For example, how does a time-triggered bus protocol affect timing in a distributed control system, and how do the timing effects change the dynamics of the system? Similarly, a functional model could be endowed with a quantity manager that measures power usage and identifies potential overloads that may result from unexpectedly synchronized interactions across a distributed system.

The notion of quantity managers brings to model-based design a capability analogous to aspect-oriented programming in software engineering [34]. Separately designed models can be woven together using quantity managers in a manner similar to the weaving of separately designed classes in aspect-oriented design.

Figure 8 shows a variant of the model of Figure 3 that has been annotated with information about the implementation. In particular, the two inputs to the fuel mover component are implemented on a network bus, which introduces a communication delay. The plot at the bottom of Figure 8 compares the fuel levels of the tanks in two cases. In the first case the service time of the bus is 0.05 and in the second case the service time is 0.15. The latter shows a clear degradation of the behavior. Between time 4 and time 5, Tank1 does not receive any fuel from Tank2 although it runs out of fuel at time 4. In a real system, this might cause the plane to crash or at least introduce severe system instability.

3.8 Semantics-Preserving Transformation and Implementation

Effective use of models requires well-defined relationships between the models and systems being modeled. In many cases, models can be used as specifications, and implementations can be synthesized from these specifications. The key challenge is that such synthesis must preserve the semantics of the implementation.

Code generation from models is a very active area of work. UML tools such as Rational Rose can generate code from executable models and templates from non-executable models. Code generators from actor modeling languages such as Simulink exist (TargetLink from dSPACE and RealTimeWorkshop from The MathWorks, for example). LabVIEW Embedded and LabVIEW FPGA are able to generate code and hardware designs, respectively, from dataflow models.

Despite considerable progress in this area, in our opinion, code generation from models is still in its infancy. The research investment so far is considerably smaller than the investment in compiler technology, despite the fact that the technical challenges are at least as great. And many problems remain only partially solved, such as modular code generation [73] and model transformation [21].

4 The State of the Art

Despite considerable progress in languages, notations, and tools, major problems persist. In practice, system integration, adaptation of existing designs, and interoperation of heterogeneous subsystems remain major stumbling blocks that cause project failures. We believe that model-based design, as widely practiced today, largely fails to benefit from the principles of platform-based design [67] as a consequence of its lack of attention to the semantics of heterogeneous subsystem composition.

Consider for example UML 2 [5, 6], widely used for modeling software systems. Its derivative SysML [61] has many features that are potentially useful for CPS. The internal block diagram notation of SysML, which is based on the UML 2 composite structure diagrams, particularly with the use of flow ports, has great potential for modeling complex systems, such as aircraft fuel systems. But it has severe weaknesses that limit

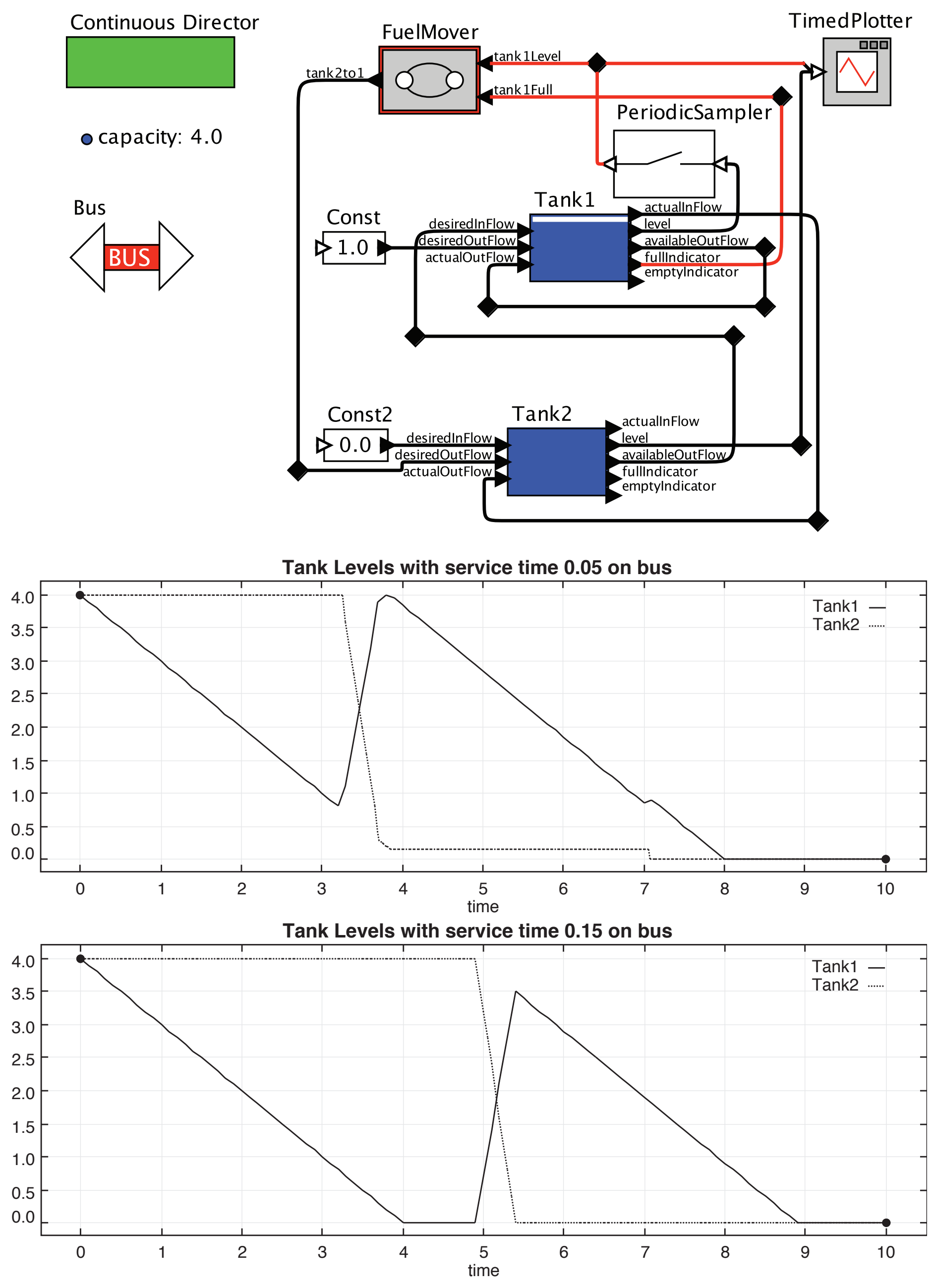


Figure 8: Fuel system model that includes some implementation detail, where the two signals into the FuelMover are transported over a single shared bus, and the transport takes time.

its ability to address system design problems. The SysML standard defines the syntax of these diagrams, not their semantics. Although the SysML standard asserts that “flow ports are intended to be used for asynchronous, broadcast, or send-and-forget interactions” [61], the standard fails to define the semantics of such interactions. Implementers of tools are free to interpret this intent, resulting in a modeling language whose semantics is defined by the tools rather than by the language itself. There are many semantic alternatives [41], consequently the same SysML diagram may be interpreted very differently by different observers.

MARTE (Modeling and Analysis of Real-Time and Embedded systems) [60] also specifically avoids “constraining” (or even defining) the execution semantics of models. Instead it focuses on providing alternative ways of describing today’s ad-hoc, non-composable design practices such as concurrency based on threads [39]. Standardizing *notation* is not sufficient to achieve effective analysis methods and unambiguous communication among designers. More importantly, without semantics, the modeling framework fails to provide a *platform* for design. Unfortunately, the very flexibility of these modeling notations may account for some of their success, because designers can become “standards compliant” without changing their existing practice. They merely have to adapt their notation.

Further weakening their semantics, UML notations can be freely reinterpreted by defining a “profile,” greatly reducing the value of the notation as an effective communication vehicle and design tool. We believe that *constraints that lead to well-defined and interoperable models have potentially far greater value*. More importantly, such constraints are essential for these modeling frameworks to become a central part of a platform-based engineering practice [68]. The challenge is to identify which constraints provide the most value while still admitting useful designs.

The inclusion by OMG of Statecharts [27] in the UML standard has helped to narrow the variability for some modeling problems, but in many cases, the exact semantics are determined by the implementation details of the supporting tools rather than by an agreed-upon standard semantics [20]. In fact, Statecharts also suffers from inadequate standardization. Despite their common origin, variants have proliferated [74]. Even the most widely used implementations of Statecharts that claim to be standards-compliant have subtle semantic differences big enough “that a model written in one formalism could be ill-formed in another formalism” [13]. In many implementations, including the widely used RHAPSODY tool from IBM, the semantics is (probably inadvertently) nondeterminate [70].

5 Conclusions

The intrinsic heterogeneity, concurrency, and sensitivity to timing of cyber-physical systems poses many modeling challenges. Much of the current work in modeling has insufficiently strong semantics to adequately address these problems. We have described some promising technologies that can help, including hybrid system modeling and simulation, concurrent and heterogeneous models of computation, the use of domain-specific ontologies to enhance modularity, and the joint modeling of functionality and implementation architectures.

References

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

[2] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems–The Polis Approach*. Kluwer, 1997.

[3] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, and Y. Watanabe. Metropolis: an integrated electronic system design environment. *Computer*, 36(4), 2003.

[4] Albert Benveniste, Benot Caillaud, Luca P. Carloni, and Alberto Sangiovanni-Vincentelli. Tag machines. In *EMSOFT*, Jersey City, New Jersey, USA, 2005. ACM.

[5] Conrad Bock. SysML and UML 2 support for activity modeling. *Systems Engineering*, 9(2):160 –185, 2006.

[6] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[7] C. Brooks, A. Cataldo, C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Zheng. HyVisual: A hybrid system visual modeler. Technical Report UCB/ERL M05/24, University of California, Berkeley, July 15 2005. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/05/hyvisual/index.htm.

[8] Christopher Brooks, Chihhong Cheng, Thomas Huining Feng, Edward A. Lee, and Reinhard von Hanxleden. Model engineering using multimodeling. In *International Workshop on Model Co-Evolution and Consistency Management (MCCM)*, Toulouse, France, 2008.

[9] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"*, 4:155–182, 1994. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/94/JEurSim/.

[10] J. R. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *International Conference on Application of Concurrency to System Design*, page 13, 2001.

[11] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. The theory of latency insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(3), 2001.

[12] Luca P. Carloni, Roberto Passerone, Alessandro Pinto, and Alberto Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1(1/2), 2006. Available from: http://dx.doi.org/10.1561/1000000001.

[13] Michelle L. Crane and Juergen Dingel. Uml vs. classical vs. rhapsody statecharts: Not all models are created equal. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume LNCS 3713, pages 97–112, Montego Bay, Jamaica, 2005. Springer.

[14] Barbara Crossette. Jet pilot who saved 304 finds heroism tainted. *New York Times*, 2001. Available from: http://query.nytimes.com/gst/fullpage.html?res=9504EFDA1738F933A2575AC0A9679C8B63.

[15] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, and Qi Zhu. A next-generation design framework for platform-based design. In *Design Verification Conference (DVCon)*, San Jose', California, 2007.

[16] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[17] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21–42, 2003. Available from: http://dx.doi.org/10.1016/S0167-6423(02)00096-5.

[18] John Eidson, Edward A. Lee, Slobodan Matic, Sanjit A. Seshia, and Jia Zou. A time-centric model for cyber-physical applications. In *Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)*, pages 21–35, 2010. Available from: http://chess.eecs.berkeley.edu/pubs/791.html.

[19] Johan Eker, Jrn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003. Available from: http://www.ptolemy.eecs.berkeley.edu/publications/papers/03/TamingHeterogeneity/.

[20] Harald Fecher, Jens Sch onborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In *International Conference on Formal Engineering Methods (ICFEM)*, volume LNCS 3785. Springer, 2005.

[21] Thomas Huining Feng and Edward A. Lee. Scalable models using model transformation. In *Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)*, 2008. Available from: http://chess.eecs.berkeley.edu/pubs/487.html.

[22] Paul A. Fishwick and Bernard P. Zeigler. A multimodel methodology for qualitative model engineering. *ACM Transactions on Modeling and Computer Simulation*, 2(1):52–81, 1992.

[23] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley, 2003.

[24] Gregor Goessler and Alberto Sangiovanni-Vincentelli. Compositional modeling in Metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Grenoble, France, 2002. Springer-Verlag.

[25] Zonghua Gu, Shige Wang, S Kodase, and K. G. Shin. An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software. In *Real-Time Systems Symposium (RTSS)*, pages 78 – 81, 2003.

[26] Cécile Hardebolle and Frédéric Boulanger. ModHel'X: A component-oriented approach to multi-formalism modeling. In *MODELS 2007 Workshop on Multi- Paradigm Modeling*, Nashville, Tennessee, USA, 2007. Elsevier Science B.V.

[27] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[28] Fernando Herrera and Eugenio Villar. A framework for embedded system specification under different models of computation in SystemC. In *Design Automation Conference (DAC)*, San Francisco, 2006. ACM.

[29] IEEE Instrumentation and Measurement Society. 1588: IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. Standard specification, IEEE, November 8 2002.

[30] Axel Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann, 2003.

[31] Juan F. Jimenez, Jose M. Giron-Sierra, C. Insaurralde, and M. Seminario. A simulation of aircraft fuel management system. *Simulation Modelling Practice and Theory*, 15:544564, 2007.

[32] Gabor Karsai, Andras Lang, and Sandeep Neema. Design patterns for open tool integration. *Software and Systems Modeling*, 4(2):157–170, 2005. Available from: `http://dx.doi.org/10.1007/s10270-004-0073-y`.

[33] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[34] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP, European Conference in Object-Oriented Programming*, volume LNCS 1241, Finland, 1997. Springer-Verlag.

[35] Edward A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999. Available from: `http://dx.doi.org/10.1023/A:1018998524196`.

[36] Edward A. Lee. Model-driven development - from object-oriented design to actor-oriented design. In *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop)*, Chicago, 2003. Available from: `http://ptolemy.eecs.berkeley.edu/publications/papers/03/MontereyWorkshopLee/`.

[37] Edward A. Lee. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, University of California, Berkeley, July 2 2003. Available from: `http://ptolemy.eecs.berkeley.edu/publications/papers/03/overview/`.

[38] Edward A. Lee. Cyber-physical systems - are computing foundations adequate? In *NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, Austin, TX, 2006. Available from: `http://ptolemy.eecs.berkeley.edu/publications/papers/06/CPSPositionPaper/`.

[39] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. Available from: `http://dx.doi.org/10.1109/MC.2006.180`.

[40] Edward A. Lee. Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC),*, pages 363 – 369, Orlando, Florida, 2008. IEEE. Available from: `http://dx.doi.org/10.1109/ISORC.2008.25`.

[41] Edward A. Lee. Disciplined heterogeneous modeling. In D. C. Petriu, N. Rouquette, and O. Haugen, editors, *Model Driven Engineering, Languages, and Systems (MODELS)*, pages 273–287. IEEE, 2010. Available from: http://chess.eecs.berkeley.edu/pubs/679.html.

[42] Edward A. Lee, Xiaojun Liu, and Stephen Neuendorffer. Classes and inheritance in actor-oriented design. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(4):29:1–29:26, 2009. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/07/classesandInheritance/index.htm.

[43] Edward A. Lee and Eleftherios Matsikoudis. The semantics of dataflow with firing. In Grard Huet, Gordon Plotkin, Jean-Jacques Lévy, and Yves Bertot, editors, *From Semantics to Computer Science: Essays in memory of Gilles Kahn*. Cambridge University Press, 2009. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/.

[44] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

[45] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 17(12):1217–1229, 1998. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/98/framework/.

[46] Edward A. Lee and Stavros Tripakis. Modal models in ptolemy. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, volume 47, pages 11–21, Oslo, Norway, 2010. Linköping University Electronic Press, Linköping University. Available from: http://chess.eecs.berkeley.edu/pubs/700.html.

[47] Edward A. Lee and Yuhong Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing Journal*, 16(3):210 – 237, 2004.

[48] Edward A. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages 25–53, Zurich, Switzerland, 2005. Springer-Verlag. Available from: http://dx.doi.org/10.1007/978-3-540-31954-2_2.

[49] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, pages 114 – 123, Salzburg, Austria, 2007. ACM. Available from: http://dx.doi.org/10.1145/1289927.1289949.

[50] Man-Kit Leung, Thomas Mandl, Edward A. Lee, Elizabeth Latronico, Charles Shelton, Stavros Tripakis, and Ben Lickly. Scalable semantic annotation using lattice-based ontologies. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Denver, CO, USA, 2009. ACM/IEEE.

[51] Jie Liu. Responsible frameworks for heterogeneous modeling and design of embedded systems. Ph.D. Thesis Technical Memorandum UCB/ERL M01/41, University of California, Berkeley, December 20 2001. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/01/responsibleFrameworks/.

[52] Jie Liu, Bicheng Wu, Xiaojun Liu, and Edward A. Lee. Interoperation of heterogeneous cad tools in Ptolemy II. In *Symposium on Design, Test, and Microfabrication of MEMS/MOEMS*, Paris, France, 1999.

[53] Xiaojun Liu and Edward A. Lee. CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, 409(1):110–125, 2008. Available from: `http://dx.doi.org/10.1016/j.tcs.2008.08.044`.

[54] Xiaojun Liu, Jie Liu, Johan Eker, and Edward A. Lee. Heterogeneous modeling and design of control systems. In Tariq Samad and Gary Balas, editors, *Software-Enabled Control: Information Technology for Dynamical Systems*. Wiley-IEEE Press, 2003.

[55] Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, pages 447–484. Springer-Verlag, 1992.

[56] I. Moir and A. Seabridge. *Aircraft Systems: Mechanical, Electrical, and Avionics Subsystems Integration*. AIAA Education Series. Wiley, third edition edition, 2008.

[57] Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation: Transactions of the Society for Modeling and Simulation International Journal of High Performance Computing Applications*, 80(9):433450, 2004.

[58] G. Nordstrom, Janos Sztipanovits, G. Karsai, and A. Ledeczi. Metamodeling - rapid design and evolution of domain-specific modeling environments. In *Proc. of Conf. on Engineering of Computer Based Systems (ECBS)*, pages 68–74, Nashville, Tennessee, 1999.

[59] Object Management Group (OMG), . System modeling language specification v1.1. Technical report, OMG, 2008. Available from: `http://www.sysmlforum.com`.

[60] Object Management Group (OMG), . A UML profile for MARTE, beta 2. OMG Adopted Specification ptc/08-06-09, OMG, August 2008. Available from: `http://www.omg.org/omgmarte/`.

[61] Object Management Group (OMG), . System modeling language specification v1.2. Standard specification, OMG, June 2010. Available from: `http://www.sysmlforum.com`.

[62] Thomas M. Parks and David Roberts. Distributed process networks in Java. In *International Parallel and Distributed Processing Symposium*, Nice, France, 2003.

[63] Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of International Conference on Computer Aided Design*, San Jose, CA., 2002.

[64] H. D. Patel and S. K. Shukla. *SystemC Kernel Extensions for Heterogeneous System Modelling*. Kluwer, 2004.

[65] Wolfgang Pree and Joseph Templ. Modeling with the timing definition language (TDL). In *Automotive Software Workshop San Diego (ASWSD) on Model-Driven Development of Reliable Automotive Services*, LNCS, San Diego, CA, 2006. Springer.

[66] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 23(1):17–32, 2004.

[67] Alberto Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, 2002.

[68] Alberto Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[69] Alberto Sangiovanni-Vincentelli, Guang Yang, Sandeep Kumar Shukla, Deepak A. Mathaikutty, and Janos Sztipanovits. Metamodeling: An emerging representation paradigm for system-level design. *IEEE Design and Test of Computers*, 2009.

[70] Wladimir Schamai, Uwe Pohlmann, Peter Fritzson, Christiaan J.J. Paredis, Philipp Helle, and Carsten Strobel. Execution of umlstate machines using modelica. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, volume 47, pages 1–10, Oslo, Norway, 2010. Linköping University Electronic Press, Linköping University. Available from: http://www.ep.liu.se/ecp/047/.

[71] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[72] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *IEEE Computer*, page 110112, 1997.

[73] Stavros Tripakis, Dai Bui, Marc Geilen, Bert Rodiers, and Edward A. Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. Technical Report UCB/EECS-2010-52,, EECS Department, University of California, Berkeley, May 7 2010.

[74] Michael von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, Lübeck, Germany, 1994. Springer-Verlag.

[75] Wikipedia. Air Transat flight 236, Retrieved February 4 2011. Available from: http://en.wikipedia.org/wiki/Air_Transat_Flight_236.

[76] Y. Xiong, E. A. Lee, X. Liu, Y. Zhao, and L. C. Zhong. The design and application of structured types in Ptolemy II. In *IEEE International Conference on Granular Computing (GrC)*, Beijing, China, 2005.

[77] Yang Zhao, Edward A. Lee, and Jie Liu. A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 259 – 268, Bellevue, WA, USA, 2007. IEEE. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/07/RTAS/.