Part I

Getting Starting

This part of this book introduces system design, modeling, and simulation. First, Chapter 1 outlines the guiding principles for disciplined modeling of heterogeneous systems. It gives a high-level overview of the models of computation that are described in detail in Part II. It also provides a highly simplified case study (an electric power system) that illustrates the roles that various models of computation play in complex system design.

Chapter 2 provides a tutorial on using Ptolemy II through its graphical user interface, Vergil. One of the goals of this book is to enable the reader to exploit the open-source Ptolemy II system to conduct experiments in system design. This chapter is intended to provide enough information to make the reader a competent user of Ptolemy II. To extend Ptolemy II, the reader is referred to Part III.

1

Heterogeneous Modeling

Edward A. Lee

Contents

Many of today's engineered systems combine heterogeneous and often complex subsystems. A modern car, for example, may combine a complex engine, electronic control units (**ECUs**), traction control systems, body electronics (for controlling windows and door locks), entertainment systems, climate control and ventilation, and a variety of safety subsystems (such as airbags). Each subsystem may be realized with a combination of software, electronics, and mechanical parts. Engineering such complex systems is quite challenging, in part because even the smallest subsystems span multiple engineering disciplines.

These complex systems also challenge the design tools that engineers use to specify, design, simulate, and analyze systems. It is no longer sufficient to sketch a mechanical structure and write down a few equations describing the interactions of the mechanical parts. Neither is it sufficient to rely entirely on software tools for 3D modeling of mechanical parts or tools for model-based design of software systems. The complex interplay across domains (mechanics, software, electronics, communication networks, chemistry, fluid dynamics, and human factors) reduce the usefulness of tools that address only a single domain.

The focus of this book is on **cyber-physical systems** (**CPS**) (Lee, 2008a, 2010a; Lee and Seshia, 2011), which combine computing and networking with physical dynamics. Cyber-physical systems require model combinations that integrate the continuous dynamics of physical processes (often described using differential equations) with models of software. Diverse models are most useful in applications where timed interactions between components are combined with conventional algorithmic computations.[1] They can also be used in traditional software systems that have concurrent[2] interactions between algorithmic components.

[1] An **algorithm** is a finite description of a sequence of steps to be taken to solve a problem. Physical processes are rarely structured as a sequence of steps; rather, they are structured as continuous interactions between concurrent components.

[2] **Concurrency**, from the Latin verb *concurrere* meaning "run together," is often taken in computer science to mean the arbitrary interleaving of two or more sequences of steps. However, this is a rather specialized interpretation of a basic concept. In this book, we take concurrency to mean simultaneous operation, with no implication of either interleaving nor sequences of steps. In particular, two continuous processes can operate concurrently without being directly representable as sequences of steps. Consider, for example, a resistive heating element immersed in a vat of water. Increasing the current through the heating element will cause the temperature of the water to rise. The electrical flow is one continuous process, as is the temperature of the water, and these processes are interacting. But neither process is reasonably representable as a sequence of steps, nor is the overall process an interleaving of such steps.

Sidebar: About the Term “Cyber-Physical Systems”

The term “cyber-physical systems” emerged around 2006, when it was coined by Helen Gill at the National Science Foundation in the United States. We are all familiar with the term “**cyberspace**,” attributed William Gibson, who used the term in the novel *Neuromancer* to refer to the medium of computer networks used for communication between humans. We may be tempted to associate the term cyberspace with CPS, but the roots of the term CPS are older and deeper. It would be more accurate to view the terms “cyberspace” and “cyber-physical systems” as stemming from the same root, “**cybernetics**,” rather than viewing one as being derived from the other.

The term “cybernetics” was coined by Norbert Wiener (Wiener, 1948), an American mathematician who had a huge impact on the development of control systems theory. During World War II, Wiener pioneered technology for the automatic aiming and firing of anti-aircraft guns. Although the mechanisms he used did not involve digital computers, the principles involved are similar to those used today in a huge variety of computer-based feedback control systems. Wiener derived the term from the Greek $\kappa\upsilon\beta\varepsilon\rho\nu\eta\tau\eta\varsigma$ (kybernetes), meaning helmsman, governor, pilot, or rudder. The metaphor is apt for control systems.

Wiener described his vision of cybernetics as the conjunction of control and communication. His notion of control was deeply rooted in closed-loop feedback, where the control logic is driven by measurements of physical processes, and in turn drives the physical processes. Even though Wiener did not use digital computers, the control logic is effectively a computation, and therefore cybernetics is the conjunction of physical processes, computation, and communication.

Wiener could not have anticipated the powerful effects of digital computation and networks. The fact that the term “cyber-physical systems” may be ambiguously interpreted as the conjunction of cyberspace with physical processes, therefore, helps to underscore the enormous impact that CPS will have. CPS leverages a phenomenal information technology that far outstrips even the wildest dreams of Wiener’s era.

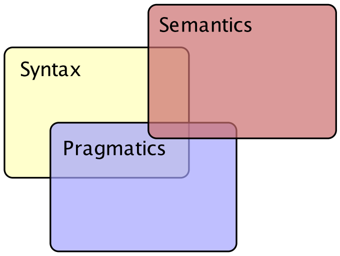


Figure 1.1: Today's design tools involve complex combinations of syntax, semantics, and pragmatics.

1.1 Syntax, Semantics, and Pragmatics

At the time of this writing, we are in the midst of a dramatic transformation in engineering tools and techniques, which are evolving to enable us to adapt to increasing system complexity and heterogeneity. In the past, entire industries were built around providing design tools for a single engineering domain, such as digital circuits, software, 3D mechanical design, and heating and ventilation systems. Today we see a growing consolidation and combination of design tools; individual tools often expand into tool suites and provide capabilities outside of their traditional domain. This evolution often entails significant growing pains, where poorly integrated capabilities yield frustratingly unexpected behaviors. Tool integration often results in "frankenware,"[3] brittle combinations of mostly incompatible tools that are extremely difficult to maintain and use effectively in combination.

Moreover, tools that have traditionally worked well within a relatively narrow domain are not as effective when used in broader domains. Today's sophisticated design tools involve complex combinations of **syntax** (how a design is represented), **semantics** (what a design means and how it works), and **pragmatics** (Fuhrmann and von Hanxleden, 2008) (how an engineer visualizes, edits, and analyzes a design). When tools are used in domains for which they were not originally designed or in combination with other tools, awkwardness may arise from incompatible syntaxes, poorly understood semantics, and inconsistent human interfaces.

[3]The term "frankenware" is due to Christopher Brooks.

Incompatibilities in syntax may arise because the structure of designs is intrinsically different (software syntax has very little in common with 3D volumes, for example). But all too often, it arises because the tools were developed in different engineering communities using different techniques. Similarly, the pragmatics of tools, such as how design files are managed and how changes are tracked, are often starkly different by historical accident. Differences in semantics are often accidental as well, sometimes arising from simple misunderstandings. Semantics may not be intuitively obvious in a different domain. A block diagram, for example, may mean something completely different to a control engineer as to a software engineer.

This book examines key concepts in heterogeneous modeling using Ptolemy II, an open-source modeling and simulation tool.[4] In contrast to most other design tools, Ptolemy II was developed from the outset to address heterogeneous systems. A key goal of the Ptolemy Project (an ongoing research effort at UC Berkeley) has been to minimize the accidental differences in syntax, semantics, and pragmatics between domains, and maximize the interoperability of designs expressed in different domains. As a consequence, Ptolemy II provides a useful laboratory for experimenting with design technologies for cyber-physical systems.

Ptolemy II integrates four distinct classes of syntaxes: block diagrams, bubble-and-arc diagrams, imperative programs, and arithmetic expressions. These syntaxes are complementary, and enable Ptolemy to address a variety of design domains. Block diagrams are used to express concurrent compositions of communicating components; bubble-and-arc diagrams are used to express sequencing of states or modes; imperative programs are used to express algorithms; and arithmetic expressions are used to express functional numeric computations.

Ptolemy II also integrates a number of semantic domains. For block diagrams, in particular, there are many distinct semantics possible. Connections between blocks represent interactions between components in a design, but what type of interaction? Is it an asynchronous message (like sending a letter)? Is it a rendezvous communication (like making a phone call)? Is it a clocked update of data (as in a synchronous digital circuit)? Does time play a role in the interaction? Is the interaction discrete or continuous? To enable heterogeneous modeling, Ptolemy II has been designed to support all of them, and is extensible to support more.

[4]Ptolemy II is available for download at `http://ptolemy.org`.

1.2 Domains and Models of Computation

A **semantic domain** in Ptolemy II, often just called a **domain**, defines the "laws of physics" for the interaction between components in a design. It provides the rules that govern concurrent execution of the components and the communication between components (such as those described above). A collection of such rules is called a **model of computation** (**MoC**). We will use the terms "model of computation" and "domain" (nearly) interchangeably, though technically we think of a domain as being an *implementation* of a MoC. The MoC is an abstract model, whereas the domain is its concrete implementation in software.

The rules that constitute a model of computation fall into three categories. The first set of rules specifies what constitutes a component. In this book, a component is generally an actor, to be defined more precisely below. The second set of rules specifies the execution and concurrency mechanisms. Are actors invoked in order? Simultaneously? Nondeterministically? The third specifies the communication mechanisms. How do actors exchange data?

Each of the MoCs discussed in this book has many possible variants, many of which have been realized in other modeling tools. In this book, we focus only on MoCs that have been realized in Ptolemy II and that have well understood and documented semantics.[5] For further context, we also provide brief descriptions and pointers to other useful MoCs that have not been realized in Ptolemy II, but have been realized in other tools.

To support the design of heterogeneous systems, Ptolemy II domains (and models of computation) interoperate with one another. This requires a level of agreement between semantic domains that is rare when tools are developed separately and then later integrated. The principles behind interoperation of domains in Ptolemy II are described in a number of papers (Eker et al., 2003; Lee et al., 2003; Goderis et al., 2009; Lee, 2010b; Tripakis et al., 2013). In this book, we focus on the practical aspects of domain interoperability, not on the theory.

Using a single, coherent software system lets us focus on domain interoperation rather than on less important incompatibilities that typically arise in tool integration. For, example, the Ptolemy II type system (which defines the types of data that can be used with various computational components) is shared by all domains, by the state machine notation,

[5]In the electronic version of this book, most illustrations of models provide a hyperlink in the caption that enables you to browse the model online. If you are reading the book on a Java-capable machine, then you can edit and execute the models shown in most of the figures.

and by the expression language. The domains are all capable of inferring and verifying appropriate data types; this functionality works seamlessly across heterogeneous models with multiple domains. Similarly, domains that include a notion of time in their semantics share a common representation of time and a (multiform) model of time. Finally, the same graphical editor spans domains, and the same XML schema is used to store design data. These agreements remove many of the practical obstacles to heterogeneous composition of models. They allow us to focus on the benefits of heterogeneous integration – most importantly, the ability to choose the domain that best matches the problem, even when the design is heterogeneous.

1.3 The Role of Models in Design

This book provides a framework for understanding and building models in Ptolemy II and, more broadly, for understanding key issues in modeling and simulating complex heterogeneous systems. This topic is broad enough that no single volume could possibly cover all of the techniques that could be useful to system designers. In this book, we focus on models that describe **dynamics**, or how a system or subsystem evolves in time. We do not cover techniques that focus primarily on the static structure of designs (such as UML class diagrams for software or 3D volumetric modeling). As a consequence, all of the models in this book are executable. We call the execution of a model a **simulation**.

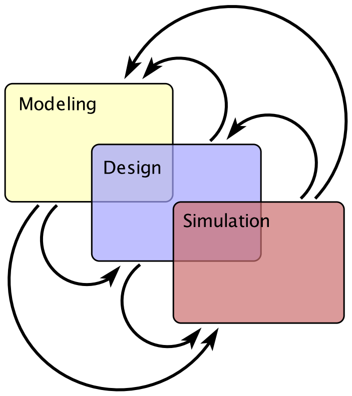


Figure 1.2: Iterative process of modeling, design, and simulation.

Figure 1.2 shows three major parts of the process of implementing systems: modeling, design, and simulation. **Modeling** is the process of gaining a deeper understanding of a system through imitation. Models imitate the system and reflect properties of the system. Models of dynamics specify *what* a system does; that is, how it reacts to stimulus from its environment, and how it evolves over time. **Design** is the structured creation of artifacts (such as software components) to implement specific functionality. It specifies *how* a system will accomplish the desired functionality. **Simulation** shows how models behave in a particular environment. Simulation is a simple form of design analysis; its goal is to lend insight into the properties of the design and to enable **testing** of the design. The models we discuss in this book can also be subjected to much more elaborate forms of analysis, including formal verification. In its most general form, **analysis** is the process of gaining a deeper understanding of a system through dissection, or partitioning into smaller, more readily analyzed pieces. It specifies *why* a system does what it does (or fails to do) what a model says it should do. We leave all analysis techniques except simulation to other texts.

As suggested in Figure 1.2, the three parts of the design process overlap, and the process iterates between them. Normally, the design process begins with modeling, where the goal is to understand the problem and to develop solution strategies.

Modeling plays a central role in modern design processes. The key principle of **model-based design** is to maximally leverage modeling to construct better designs. To be effective, models must be reasonably faithful, of course, but they also must be understandable and analyzable. To be understandable and analyzable, a model needs to have a clear meaning (a clear semantics).

Models are expressed in some **modeling language**. For example, a procedure may be expressed in Java or in C, so these programming languages are in fact modeling languages for procedures. A modeling language has a **strong semantics** if models expressed in the language have a clear and unambiguous meaning. Java, for example, has a stronger semantics than C, as illustrated by the following example.

Example 1.1: Suppose that the arguments to a procedure are of type *int*. In Java, this data type is well defined, but not in C. In C, *int* may represent a 16-bit integer or a 32-bit integer, for example. The behavior of the procedure may be quite different depending on which implementation is provided. Particularly, overflow occurs more easily with 16-bit integers than with 32-bit integers.

Sidebar: Models vs. Realizations of Systems

Models must be used with caution. The **Kopetz principle** (named after Prof. Dr. Hermann Kopetz of TU-Vienna, who taught us this principle), paraphrased, is: *Many properties that we assert about systems (determinism, timeliness, reliability) are in fact not properties of an implemented system, but rather properties of a model of the system.*

Golomb (1971) emphasizes understanding the distinction between a model and thing being modeled, famously stating "you will never strike oil by drilling through the map!" In no way, however, does this diminish the value of a map! Consider **determinism**. A model is **determinate** if it produces a uniquely defined output for each particular input. It is **nondeterminate** if there are multiple possible outputs for any particular input. Although this seems like a simple definition, there are many subtleties. What do we mean by a "particular input?" Does the time at which the input arrives matter? What do we mean by a "uniquely defined output?" Should we consider how the system behaves when its implementation hardware fails?

Any statement about the determinism of a physical "implemented" system is fundamentally a religious or philosophical assertion, not a scientific one. We may assert that no real physical system is determinate. How will it behave when it is crushed, for example? Or we may conversely assert that everything in the physical world is preordained, a concept that we find farfetched, difficult to refute, and not very useful.

For models, however, we can make definitive assertions about their determinism. For example, a procedure defined in a programming language may be determinate in that the returned value of the procedure depends only on the arguments. No actual realization of the procedure is actually determinate in an absolute sense (the hardware may fail and no returned value will be produced at all). The procedure is a **model** defined within a **formal framework** (the semantics of the language). It models the execution of a machine abstractly, omitting information. The time at which the inputs are provided makes no difference to the model, so time is not part of what we mean by a "particular input." The inputs and outputs are just data, and the procedure defines the relationship between the inputs and outputs. This point about models is supported by Box and Draper (1987), who state "Essentially, all models are wrong, but some are useful." The usefulness of a model depends on the model **fidelity**, the degree to which a model accurately imitates the system being modeled. But models are *always* an approximation.

Many popular modeling languages based on block diagrams have quite **weak semantics**. It is common, for example, for modeling languages to adopt a block diagram notation without giving precise meaning to the lines drawn between blocks; they vaguely represent the fact that components interact. (For examples, see the sidebar on page 14.) Modeling languages with weak semantics are harder to analyze. Their value lies instead in their ability to informally communicate design concepts among humans.

1.4 Actor Models

Ptolemy II is based on a class of models called **actor-oriented models**, or more simply, **actor models**. **Actors** are components that execute concurrently and share data with each other by sending messages via ports.

Example 1.2: Consider, for example, the Ptolemy model shown in Figure 1.3. This model shows three actors, each of which has one port. Actor A sends messages to actors B and C via its port (the Relation diamond indicates that the output from A goes to both B and C).

The sum of all of the messages communicated via a port is referred to as a **signal**. The **Director** block in the example specifies the domain (and hence the model of computation). Most of this book is devoted to explaining the various domains that have been realized in Ptolemy II.

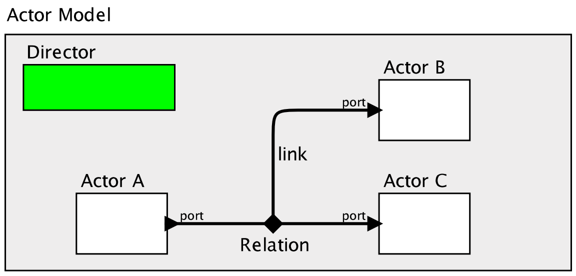


Figure 1.3: Visual rendition of a simple actor model.

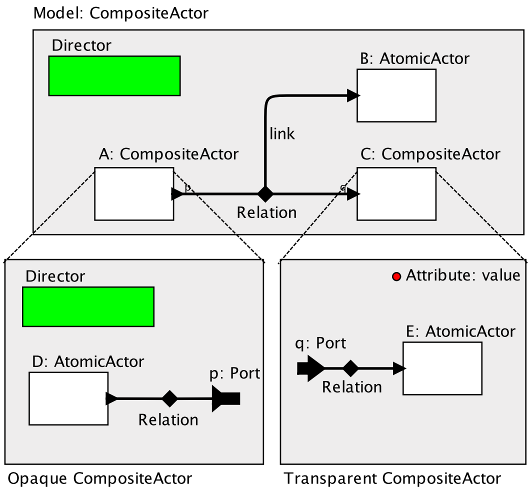


Figure 1.4: A hierarchical actor model consisting of a top-level composite actor and two submodels, each of which is also a composite actor.

1.5 Model Hierarchy

Models of complex systems are often complex. There is an art (the art of **model engineering**) to constructing good models of complex systems. A good model of a complex system provides relatively simple views of the system in order to faciliate easier understanding and analysis. A key approach to creating models with simplified views is to use modeling **hierarchy**, where what appears to be single component in one model is, internally, another model.

A hierarchical actor model is shown in Figure 1.4. It is an elaboration of Figure 1.3 where actors A and C are revealed to be themselves actor models. An **atomic actor** (where **atomic** comes from the ancient Greek **atomos**, meaning indivisible), is one that is not internally defined as an actor model. A **composite actor**, in contrast, is itself a composition of other actors. The ports *p* and *q* in the figure bridge the levels of hierarchy. A communication from D, for example, will eventually arrive at actor E after traversing ports and levels of the hierarchy.

1.6 Approaches to Heterogeneous Modeling

There are many approaches to heterogeneous modeling (Brooks et al., 2008). In **multi-view modeling**, distinct and separate models of the same system are constructed to model different aspects of a system. For example, one model may describe dynamic behavior, while another describes physical design and packaging. In **amorphous heterogeneity**, distinct modeling styles are combined in arbitrary ways within the same model without the benefit of structure. For example, some component interactions in a model may use rendezvous messaging (where both a sender and a receiver must be ready before a communication can occur), while others use asynchronous message passing (where the receiver receives the communication at some indeterminate time after the sender sends it). In **hierarchical multimodeling**, hierarchical compositions of distinct modeling styles are combined to take advantage of the unique capabilities and expressiveness of each style.

Sidebar: About the Term "Actors"

Our notion of actor-oriented modeling is related to the term "actor" as introduced in the 1970's by Hewitt to describe the concept of autonomous reasoning agents (Hewitt, 1977). The term evolved through the work of Agha and others to describe a formalized model of concurrency (Agha et al., 1997). Agha's actors each have an independent thread of control and communicate via asynchronous message passing. The term "actor" was also used in Dennis's dataflow models (Dennis, 1974) of discrete atomic computations that react to the availability of inputs by producing outputs sent to other actors.

In this book, the term "actor" embraces a larger family of models of concurrency. They are often more constrained than general message passing and do not necessarily conform with a dataflow semantics. Our actors are still conceptually concurrent, but unlike Agha's actors, they need not have their own thread of control. Unlike Dennis' actors, they need not be triggered by input data. Moreover, although communication is still achieved through some form of message passing, it need not be asynchronous.

Actors are *components* in systems and can be compared to **objects**, software components in object-oriented design. In prevailing object-oriented languages (such as Java, C++, and C#), the interfaces to objects are primarily **methods**, which are procedures that modify or observe the state of objects. By contrast, the actor interfaces are primarily **ports**, which send and receive data. They do not imply the same sequential transfer of control that procedures do, and hence they are better suited to concurrent models.

Sidebar: Actors in UML, SysML, and MARTE

The **Object Management Group** (**OMG**) has standardized a number of notations that relate strongly to the block diagram syntax common in actor models. The actor models in this book relate to **composite structure diagrams** of UML 2 (the second version of the **unified modeling language**) (Bock, 2006; Booch et al., 1998), or more directly its derivative **SysML** (Object Management Group (OMG), 2008a). The **internal block diagram** notation of SysML, particularly with the use of flow ports, is closely related to actor models. In SysML, the actors are called “blocks.” (The term “actor” is used in UML for another purpose.)

SysML, however, emphasizes how model diagrams are rendered (their visual syntax), and leaves many details open about what the diagrams mean and how the models operate (their semantics). For example, although the SysML declares that “flow ports are intended to be used for asynchronous, broadcast, or send-and-forget interactions” (Object Management Group (OMG), 2008a), there is nothing like an MoC in SysML. Different SysML tools may give different behavior to flow ports and still be compliant with the standard. A single SysML model may represent multiple designs, and the behavior of the model may depend on the tools used to interpret the model. The emphasis of SysML is on standardizing the notation, not the meaning of the notation.

In contrast, the emphasis in Ptolemy II is on the semantics of models, rather than on how they are rendered (visually or otherwise). The visual notation is incidental, and in fact is not the only representation for a Ptolemy II model. Ptolemy II directors give models a very specific meaning. This concrete meaning ensures that a model means the same thing to different observers, and enables interoperation of heterogeneous models.

MARTE (modeling and analysis of real-time and embedded systems) puts more emphasis than SysML on the behavior of models (Object Management Group (OMG), 2008b). It avoids “constraining” the execution semantics, making the standard flexible, enabling representation of many prevalent real-time modeling techniques. In contrast, the emphasis in Ptolemy II is less on capturing existing design practices, and more on providing precise and well-defined models of system behavior. MARTE, interestingly, includes a multiform time model (André et al., 2007) not unlike that supported by Ptolemy II.

An example familiar to software engineers is Statecharts (Harel, 1987), which hierarchically combines synchronous concurrent composition with finite state machines. Another example of hierarchical modeling is **cosimulation**, where two distinct simulation tools are combined using a standardized interface such as the Simulink S function interface or the functional mockup interface (**FMI**) from the Modelica Association.[6] It is also possible to support heterogeneous modeling by creating very flexible or underspecified modeling frameworks that can be adapted to cover the models of interest. The downside of this approach is weak semantics. The goal of Ptolemy II is to achieve strong semantics, yet embrace heterogeneity and provide mechanisms for heterogeneous models to interact concurrently.

As shown in Figure 1.4, one can partition a complex model into a hierarchical tree of nested submodels. At each level, the submodels can be joined together to form a network of interacting actors. Ptolemy II constrains each level of the hierarchy to be locally homogeneous, using a common model of computation. These homogeneous networks can then be hierarchically combined to form a larger heterogeneous model. The advantage to this approach is that each part of the system can be modeled using the model of computation that provides the best match for its processing requirements — yet each model of computation provides strong semantics to ensure that it is relatively easy to understand, analyze, and execute.

In Ptolemy II, a director defines the semantics of a model. In Figure 1.4, there are two directors. The one at the top level defines the interaction between actors A, B, and C. Since C does not internally contain a director, the same top-level director governs the interactions with actor E. Actor C is called a **transparent composite actor**; its contained model is visible to its director.

In contrast, actor A internally contains another director. That inside director governs the interaction of actors within the sub model (in this simple example, there is only one such actor, but there could be more). Actor A is called an **opaque composite actor**, and its contents are hidden from A's outside director. To distinguish the two directors, we call the outside director the **executive director**. To the executive director, actor A looks just like an atomic actor. But internally, A contains another model.

The directors at different levels of the hierarchy need not implement the same MoC. Opaque composite actors, therefore, are Ptolemy's way of realizing hierarchical multimodeling and cosimulation.

[6] `http://www.functional-mockup-interface.org`

Sidebar: Plurality of Models

Occam's razor is a principle in science and engineering that encourages selection of those theories and hypotheses that require the fewest assumptions, postulates, or entities to explain a given phenomenon. The principle can be expressed as "entities must not be multiplied beyond necessity" (*entia non sunt multiplicanda praeter necessitatem*) or as "plurality should not be posited without necessity" (*pluralitas non est ponenda sine necessitate*) (Encyclopedia Britannica, 2010). The principle is attributed to 14th-century English logician, theologian and Franciscan friar William of Ockham.

Despite its compelling value, the principle has limitations. Immanuel Kant, for example, felt a need to moderate the effects of Occam's razor, stating "the variety of beings should not rashly be diminished." (*entium varietates non temere esse minuendas*) (Smith, 1929). Einstein allegedly remarked, "everything should be made as simple as possible, but not simpler" (Shapiro, 2006).

When applied to design techniques, Occam's razor biases us towards using fewer and simpler design languages and notations. However, experience indicates that both redundancy and diversity can be beneficial. For example, there is benefit to using UML class diagrams even if the information they represent is already encoded in a C++ program. There is also value in UML use-case diagrams, which express concepts that are not encoded in the C++ program and are also not (directly) represented in the UML class diagram. The three representations serve different purposes, though they represent the same underlying process.

The fact that many different notations are used in UML and its derivatives runs counter to the principle in Occam's razor. Ironically, the unified modeling language (**UML**) originated in the 1990s to *reduce* the diversity of notations used to express object-oriented software architectures (Booch et al., 1998). So what is gained by this anti-razor?

Design of software systems is essentially a creative process; engineers create programs that did not previously exist. Occam's razor should be applied only cautiously to creative processes, because creativity often flourishes when there are multiple media with which to achieve the desired effect. UML facilitates the creative process by offering more abstract notations than C++ source code, and these notations encourage experimentation with design and communication of design ideas.

Sidebar: About Heterogeneous Models

Some authors use the term **multi-paradigm modeling** to describe approaches that mix models of computation (Mosterman and Vangheluwe, 2004). Ptolemy II focuses on techniques that combine actors with multi-paradigm modeling. An early systematic approach to such mixed models was realized in Ptolemy Classic (Buck et al., 1994), the predecessor to Ptolemy II (Eker et al., 2003). Influenced by the Ptolemy approach, SystemC is capable of realizing multiple MoCs (Patel and Shukla, 2004; Herrera and Villar, 2006). So are ModHel'X (Hardebolle and Boulanger, 2007) and ForSyDe (Jantsch, 2003; Sander and Jantsch, 2004).

Another approach supports mixing concurrency and communication mechanisms without the structural constraints of hierarchy (Goessler and Sangiovanni-Vincentelli, 2002; Basu et al., 2006). A number of other researchers have tackled the problem of heterogeneity in creative ways (Burch et al., 2001; Feredj et al., 2009).

It is also possible to use **tool integration**, where different modeling tools are combined either through interchange languages or through co-simulation (Liu et al., 1999; University of Pennsylvania MoBIES team, 2002; Gu et al., 2003; Karsai et al., 2005). This approach is challenging, however, and yields fragile tool chains. Many tools lack documentation on how and where they can be extended to enable cross-tool integration; implementing and maintaining integration requires considerable effort. Challenges include API incompatibilities, unstable or undocumented APIs, unclear semantics, syntactic incompatibilities, and unmaintainable code bases.Tool integration proves to be a painful way to accomplish heterogeneous design. A better approach is to focus on the semantics of interoperation, rather than on the software problems of tool integration. Good software architectures for interoperation will emerge only from a good understanding of the semantics of interoperation.

In Ptolemy, each model contains a director that specifies the MoC being used and provides either a code generator or an interpreter for the MoC (or both). An interesting alternative is given by "42" (Maraninchi and Bhouhadiba, 2007), which integrates a custom MoC with the model.

Sidebar: Tools Supporting Heterogeneous Models

Several widely used tools provide fixed combinations of a few MoCs. Commercial tools include Simulink/StateFlow (from The MathWorks), which combines continuous- and discrete-time actor models with finite-state machines, and LabVIEW (from National Instruments), which combines dataflow actor models with finite-state machines and a time-driven MoC. Statemate (Harel et al., 1990) and SCADE (Berry, 2003) combine finite-state machines with a synchronous/reactive formalism (Benveniste and Berry, 1991). Giotto (Henzinger et al., 2001) and TDL (Pree and Templ, 2006) combine FSMs with a time-driven MoC. Several hybrid system modeling and simulation tools combine continuous-time dynamical systems with FSMs (Carloni et al., 2006).

The Y-chart approach supports heterogeneous modeling and is popular for hardware-software codesign (Kienhuis et al., 2001). This approach separates modeling of the hardware implementation from modeling of application behavior (a form of multi-view modeling), and provides mechanisms for bringing these disparate models together. These mechanisms allow developers to trade off hardware cost and complexity with software design. Metropolis is a particularly elegant tool for this purpose (Goessler and Sangiovanni-Vincentelli, 2002). It introduces a "quantity manager" that mediates interactions between the desired functionality and the resources required to implement that functionality.

Modelica (Fritzson, 2003; Modelica Association, 2009) also has actor-like semantics in the sense that components are concurrent and communicate via ports, but the ports are neither inputs nor outputs. Instead, the connections between ports declare equation constraints on variables. This approach has significant advantages, particularly for specifying physical models based on differential-algebraic equations (**DAE**s). However, the approach also appears to be harder to combine heterogeneously with other MoCs.

DESTECS (design support and tooling for embedded control software) is a tool supported by a consortium from academia and industry that has a focus on fault-tolerant embedded systems (Fitzgerald et al., 2010). This tool integrates continuous-time models made in 20-sim (Broenink, 1997) and discrete-event models in VDM (Vienna Development Method) (Fitzgerald et al., 2008). DESTECS synchronizes time and passes variables between the two tools.

1.7 Models of Time

Some models of computation have a notion of **time**. Specifically, this means that communication between actors and computation performed by actors occurs on a logical time line. Even more specifically, this means that there is a notion of two actions (communication or computation) being either ordered in time (one occurs before the other) or being simultaneous. A notion of time may also have a metric, meaning (loosely) that the time gap between two actions may be measured.

A key mechanism that Ptolemy II provides for interoperability of domains is a coherent notion of time. This mechanism has proven effective even for combining models of computation that have no notion of time (such as dataflow models and finite state machines), with models of computation that depend strongly on time (such as discrete-event models and continuous-time models). In this section, we outline key features of this mechanism.

1.7.1 Hierarchical Time

The model hierarchy discussed in Sections 1.5 and 1.6 is central to the management of time. Typically, only the top-level director advances time. Other directors in a model obtain the current model time from their enclosing director. If the top-level director does not implement a timed model of computation, then time does not advance. Hence, timed models always contain a top-level director that implements a timed model of computation.

Timed and untimed models of computation may be interleaved in the hierarchy. As we will discuss later, however, there are certain combinations that do not make sense, while other combinations are particularly useful, particularly the modal models discussed in Chapter 8.

Time can also advance non-uniformly in a model. In the modal models of Chapter 8, the advancement of time can be temporarily suspended in a submodel (Lee and Tripakis, 2010). More generally, as explained in Chapter 10, time may also progress at different rates at different levels of the hierarchy. This feature is particularly useful for modeling distributed systems where maintaining a perfectly coherent uniform time base is not physically possible. It is referred to as **multiform time**, and it enables highly realistic models that explicitly recognize that time can only be imperfectly measured.

1.7.2 Superdense Time

In addition to providing multiform time, Ptolemy II provides a model of time known as **superdense time** (Manna and Pnueli, 1993; Maler et al., 1992; Lee and Zheng, 2005; Cataldo et al., 2006). A superdense time value is a pair (t, n), called a **time stamp**, where t is the **model time** and n is a **microstep** (also called an **index**). The model time represents the time at which some event occurs, and the microstep represents the sequencing of events that occur at the same model time. Two time stamps (t, n_1) and (t, n_2) can be interpreted as being **simultaneous** (in a weak sense) even if $n_1 \neq n_2$. A stronger notion of **simultaneity** would require the time stamps to be equal (both in model time and microstep). An example illustrates the value of superdense time.

Example 1.3: To understand the role of the microstep, consider Newton's cradle, a toy with five steel balls suspended by strings, shown in Figure 1.5. If you lift the first ball and release it, it strikes the second ball, which does not move. Instead, the fifth ball reacts by rising.

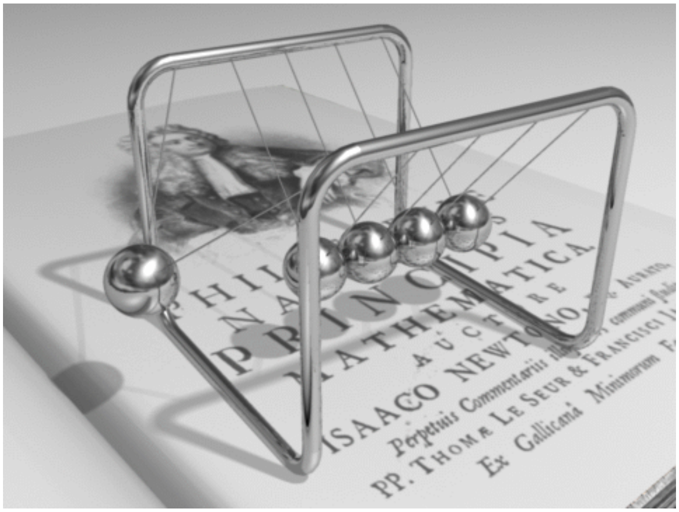

PHILOSOPHIÆ NATURALIS
PRINCIPIA
MATHEMATICA.
AUCTORE
ISAACO NEWTONO.
Perpetuis Commentariis illustrata, communi studio
PP. THOMÆ LE SEUR & FRANCISCI JACQUIER
Ex Gallicanâ Minimorum Familiâ

Figure 1.5: Newton's cradle. Image by Dominique Toussaint, made available under the terms of the GNU Free Documentation License, Version 1.2 or later.

Consider the momentum p of the second ball as a function of time. The second ball does not move, so its momentum must be everywhere zero. But the momentum of the first ball is somehow transferred to the fifth ball, passing through the second ball. So the momentum cannot be always zero.

Let $\mathbb{R}$ represent the real numbers. Let $p\colon \mathbb{R} \to \mathbb{R}$ be a function that represents the momentum of this second ball, and let τ be the time of the collision. Then

$$p(t) = \begin{cases} P & \text{if } t = \tau \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

for some constant P and for all $t \in \mathbb{R}$. Before and after the instant of time τ, the momentum of the ball is zero, but at time τ, it is not zero. Momentum is proportional to velocity, so

$$p(t) = Mv(t),$$

where M is the mass of the ball. Hence, combining with (1.1),

$$v(t) = \begin{cases} P/M & \text{if } t = \tau \\ 0 & \text{otherwise.} \end{cases} \tag{1.2}$$

The position of a mass is the integral of its velocity,

$$x(t) = x(0) + \int_0^t v(\tau)d\tau,$$

where $x(0)$ is the initial position. The integral of the function given by (1.2) is zero at all t, so the ball does not move, despite having a non-zero momentum at an instant.

The above physical model mostly works to describes the physics, but has two flaws. First, it violates the basic physical principle of conservation of momentum. At the time of the collision, all three middle balls will simultaneously have non-zero momentum, so seemingly, aggregate momentum has magically increased. Second, the model cannot be directly converted into a discrete representation.

A **discrete** representation of a signal is a sequence of values that are ordered in time (for mathematical details, see the sidebar on page 334). Any such representation of the momentum in (1.1) or velocity in (1.2) is ambiguous. If the sequence does not include the value at the time of the collision, then the representation does not capture the fact that momentum is transferred through the ball. If the representation does include the value at the time of the collision, then the representation is indistinguishable from a representation of a signal that has a non-zero momentum over some interval of time, and therefore models a ball that does move. In such a discrete representation, there is no semantic distinction between an instantaneous event and a rapidly varying continuous event.

Superdense time solves both problems. Specifically, the momentum of the second ball can be unambiguously represented by a sequence of samples where $p(\tau, 0) = 0$, $p(\tau, 1) = P$, and $p(\tau, 2) = 0$, where τ is the time of the collision. The third ball has non-zero momentum only at superdense time $(\tau, 2)$. At the time of the collision, each ball first has zero momentum, then non-zero, then zero again, all in an instant. The event of having non-zero momentum is weakly simultaneous for all three middle balls, but not strongly simultaneous. Momentum is conserved, and the model is unambiguously discrete.

One could argue that the physical system is not actually discrete. Even well-made steel balls will compress, so the collision is actually a continuous process, not a discrete event. This is true, but when building models, we do not want the modeling formalism to force us to construct models that are more detailed than is appropriate. Such a model of Newton's cradle would be far more sophisticated, and the resulting non-linear dynamics would be far more difficult to analyze. The fidelity of the model would improve, but at a steep price in understandability and analyzability.

The above example shows that physical processes that include instantaneous events are better modeled using functions of the form $p\colon \mathbb{R} \times \mathbb{N} \to \mathbb{R}$, where $\mathbb{N}$ represents the natural numbers, rather than the more conventional $p\colon \mathbb{R} \to \mathbb{R}$. The latter is adequate for continuous processes, but not for discrete events. At any time $t \in \mathbb{R}$, the signal p has a sequence of values, ordered by their microsteps. This signal cannot be misinterpreted as a rapidly varying continuous signal.

We say that two time stamps (t_1, n_1) and (t_2, n_2) are **weakly simultaneous** if $t_1 = t_2$, and **strongly simultaneous** if, in addition, $n_1 = n_2$.

Thus we can represent causally-related, but weakly simultaneous events. A signal may have two *distinct* events at with time stamps (t, n_1) and (t, n_2), where $n_1 \neq n_2$. A signal may therefore include weakly simultaneous, but distinct, events. Two distinct signals may contain strongly simultaneous events, but a single signal cannot contain two distinct strongly simultaneous events. This model of time unambiguously represents discrete events, discontinuities in continuous-time signals, and sequences of zero-time events in discrete signals.

Superdense time is ordered lexicographically (like a dictionary), which means that $(t_1, n_1) < (t_2, n_2)$ if either $t_1 < t_2$, or $t_1 = t_2$ and $n_1 < n_2$. Thus, an event is considered to occur before another if its model time is less or, if the model times are the same, if its microstep is lower. Time stamps are a particular realization of **tags** in the tagged-signal model of Lee and Sangiovanni-Vincentelli (1998).

1.7.3 Numeric Representation of Time

Computers cannot perfectly represent real numbers, so a time stamp of form $(t, n) \in \mathbb{R} \times \mathbb{N}$ is not realizable. Many software systems approximate a time t using a double-precision floating point number. But such a representation has two serious disadvantages. First, the **precision** of a number (how close it is to the next smaller or large representable number) depends on its magnitude. Thus, as time increases in such systems, the precision with which time is represented decreases. Second, addition and subtraction can introduce quantization errors in such a representation, so it is not necessarily true that $(t_1+t_2)+t_3 = t_1 + (t_2 + t_3)$. This significantly weakens the semantic notion of simultaneity, since whether two events are (weakly or strongly) simultaneous may depend on how their time stamps were computed.

Ptolemy II solves this problem by making the **time resolution** a single, global constant. Model time is given as $t = mr$, where m is an arbitrarily large integer, and the time resolution r is a double-precision floating point number. The multiple m is realized as a Java BigInteger (an arbitrarily large integer), so it will never overflow. The time resolution r, a *double*, is a parameter shared by all the directors in a model. A model, therefore, has the same time resolution throughout its hierarchy and throughout its execution, no matter how big time gets. Moreover, addition and subtraction of time values does not suffer quantization errors. By default, the time resolution is $r = 10^{-10}$, which may represent one tenth of a nanosecond. Then, for example, $m = 10^{11}$ represents 10 seconds.

In Ptolemy II, the microstep n in a time stamp (t, n) is represented as an *int*, a 32-bit integer. The microstep, therefore, is vulnerable to overflow. Such overflow may be prevented by avoiding models that have chattering Zeno behavior, as discussed in Chapter 7.

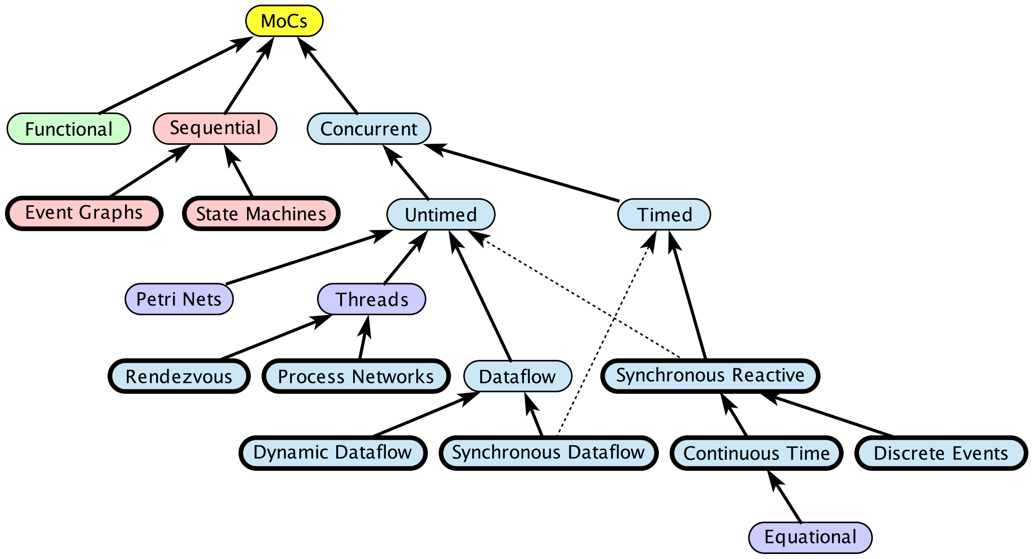


Figure 1.6: Summary of the relationship between models of computation. The ones with bold outlines are covered in detail in this Book.

1.8 Overview of Domains and Directors

In Ptolemy II an implementation of a model of computation is called a domain.[7] In this section, we briefly describe domains that have been realized in Ptolemy II. This is not a complete list; the intent is to show the diversity of the models of computation under study. These and other domains are described in subsequent chapters in more detail. Figure 1.6 summarizes the relationships between these domains.

All of the domains discussed here ensure determinism unless the model explicitly specifies nondeterministic behavior. That is, nondeterminism, if desired, must be explicitly built into the models; it does not arise accidentally from weak semantics in the modeling framework. A domain is said to be determinate if the signals sent between actors —

[7]The 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 it governs.

including data values carried by messages, their order, and their time stamps — do not depend on arbitrary scheduling decisions, despite the concurrency in the model. Ensuring determinism is far from trivial in concurrent MoCs, and providing reasonable nondeterminate mechanisms is also challenging. The goal is that, when a model includes nondeterminate behavior, it should be explicitly specified by the builder of the model; it should not appear accidentally, nor should it surprise the user.

Dataflow. Ptolemy II includes several dataflow domains, described in Chapter 3. The execution of an actor in dataflow domains consists of a sequence of **firings**, where each firing occurs as a reaction to the availability of input data. A firing is a (typically small) computation that consumes the input data and produces output data.

The synchronous dataflow (SDF) domain (Lee and Messerschmitt, 1987b) is particularly simple, and is possibly the most used domain of all. When an actor is executed in SDF, it consumes a fixed amount of data from each input port, and produces a fixed amount of data to each output port. An advantage of the SDF domain is that (as described in Chapter 3) the potential for deadlock and boundedness can be statically checked, and schedules (including parallel schedules) can be statically computed. Communication in this domain is realized with **first-in, first-out** (**FIFO**) queues with fixed finite capacity, and the execution order of components is statically scheduled. SDF can be timed or untimed, though it is usually untimed, as suggested in Figure 1.6.

In contrast, the dynamic dataflow (DDF) domain is more flexible than SDF and computes schedules on the fly. In DDF, the capacity of the FIFO queues is not bounded. DDF is useful when communication patterns between actors are dependent on the data that is passed between actors.

Dataflow models are ideal for representing **streaming** systems, where sequences of data values flow in relatively regular patterns between components. Signal processing systems, such as audio and video systems, for example, are a particularly good match.

Process Networks. In the process network (PN) domain, described in Chapter 4, actors represent concurrent processes that communicate by (conceptually infinite capacity) FIFO queues (Lee and Parks, 1995). Writing to the queues always succeeds immediately, while reading from an empty queue blocks the reader process. The simple blocking-read, nonblocking-write strategy ensures the determinacy of the model (Kahn and MacQueen, 1977). Nevertheless, we have extended the model to support certain forms of nondeterminism. Each actor executes in its own Java thread, so on multicore machines they can execute in parallel. This domain is untimed. The PN domain realizes a generalization of dataflow where instead of discrete firings, actors represent continually executing processes (Lee and Matsikoudis, 2009).

PN is suitable for describing concurrent processes that communicate asynchronously by sending messages to one another. Messages are eventually delivered in the same order they are sent. Message delivery is presumed to be reliable, so the sender does not expect nor receive any confirmation. This domain has a "send and forget" flavor.

PN also provides a relatively easy way to get parallel execution of models. Each actor executes in its own thread, and most modern operating systems will automatically map threads onto available cores. Note that if the actors are relatively fine-grained, meaning that they perform little computation for each communication, then the overhead of multithreading and inter-thread communication may overwhelm the performance advantages of parallel execution. Thus, model builders should expect performance advantages only for coarse-grained models.

Rendezvous. The Rendezvous domain, also described in Chapter 4, is similar to PN in that actors represent concurrent processes. However, unlike PN's "send and forget" semantics, in the Rendezvous domain, actors communicate by atomic instantaneous data exchanges. When one actor sends data to another, the sender will block until the receiver is ready to receive. Similarly, when one actor attempts to read input data, it will block until the sender of the data is ready to send the data. As a consequence, the process that first reaches a rendezvous point will stall until the other process reaches the same rendezvous point (Hoare, 1978). It is also possible in this domain to create multi-way rendezvous, where several processes must all reach the rendezvous point before any process can continue. Like PN, this domain is untimed, supports explicit nondeterminism, and can transparently leverage multicore machines.

The Rendezvous domain is particularly useful for modeling asynchronous resource contention problems, where a single resource is shared by multiple asynchronous processes.

Synchronous-Reactive. The synchronous-reactive (SR) domain, described in Chapter 5, is based on the semantics of synchronous languages (Benveniste and Berry, 1991; Halbwachs et al., 1991; Edwards and Lee, 2003a). The principle behind synchronous languages is simple, although the consequences are profound. Execution follows "ticks" of a global "clock." At each tick, each variable (represented visually in Ptolemy II by the wires that connect the blocks) may or may not have a value. Its value (or absence of

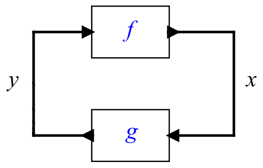


Figure 1.7: A simple feedback system.

value) is given by an actor whose output port is connected to the wire. The actor realizes a function that maps the values at its input ports to the values at its output ports (the function can vary from tick to tick). For example, in Figure 1.7, the variables x and y at a particular tick are related by

$$x = f(y), \text{ and } y = g(x).$$

The task of the domain's director is to find, at each tick, values of x and y that solve these equations. This solution is called a **fixed point**. The SR domain is by default untimed, but it can optionally be timed, in which case there is a fixed time interval between ticks.[8]

The SR domain is similar to dataflow and PN in that actors send streams of data to one other. Unlike dataflow, however, the streams are synchronized; at a tick of the clock, every communication path either has a message, or the message is unambiguously absent. Dataflow models, by contrast, are more asynchronous; a message may be "absent" simply because it hasn't arrived yet due to an accident of scheduling. To prevent nondeterminism, PN and dataflow have no semantic notion of an "absent" input. Inputs always have messages (or will have messages, in which case the actor is required to wait for the messages to arrive).

SR is well suited to situations with more complex control flow, where an actor may take different actions depending on whether a message is present or not. By synchronizing actions, the domain handles these scenarios without nondeterminism. SR is less concurrent than dataflow or PN, since each tick of the clock must be tightly orchestrated. As a consequence, it is harder to execute in parallel.

Finite-State Machine. The finite state machine (FSM) domain, described in Chapter 6, is the only domain discussed here that is not concurrent. The components in this domain

[8]If you need a variable time interval between ticks, you can accomplish this by placing an SR model within a DE model.

are not actors, but rather represent states, and the relations represent not communication paths, but rather transitions between states. Transitions have guards that determine when state transitions occur.

An FSM can be used to define the behavior of an actor used in any of the other domains. The actor can have any number of input and output ports. When that actor executes, the FSM reads the inputs, evaluates the guards to determine which transition to take, and produces outputs as specified on the selected transition. FSMs can also have local variables whose values can be modified by transitions (providing a model of computation that is known as an extended state machine).

An FSM can also be used to create a rich class of hierarchical models known as modal models, discussed in Chapter 8 (Lee and Tripakis, 2010). In a modal model, states of an FSM contain submodels that process inputs and produce outputs. Each state of the FSM represents a mode of execution, and the mode refinement defines the behavior in that mode. The mode refinement is a submodel with its own director that is active only when the FSM is in the corresponding state. When a submodel is not active, its local time does not advance, as explained above in Section 1.7.1.

Discrete Event. In the discrete-event (DE) domain, described in Chapter 7, actors communicate through events placed on a time line. Each event has a value and a time stamp, and 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, actors in DE are causal.

The execution of this model uses a global event queue. When an actor generates an output event, the event is slotted into the queue according to its time stamp. 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. The DE domain supports simultaneous events. At each time where at least one actor fires, the director computes a fixed point, similar to SR (Lee and Zheng, 2007). DE is closely related to the well-known DEVS (discrete event system specification) formalism (Zeigler et al., 2000), which is widely used for simulating large, complex systems. The semantics of the Ptolemy II variant of DE is given by Lee (1999).

DE is well suited for modeling the behavior of complex systems over time. It can model networks, digital hardware, financial systems, and human organizational systems, for example. Chapter 10 shows how DE can be extended to leverage multiform time.

Continuous time. The Continuous time domain (Lee and Zheng, 2005), described in Chapter 9, models ordinary differential equations (ODEs), while also supporting 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.

The Continuous model computes solutions to ODEs using numerical methods. As with SR and DE, at each instant, the director computes a fixed point for all signal values (Lee and Zheng, 2007). In each iteration, time is advanced by an amount determined by the ODE solver. To advance time, the director chooses a time stamp with the help of a solver and speculatively executes actors through this time step. If the time step is sufficiently small (key events such as level crossings, mode changes, or requested firing times are not skipped over, and the numerical integration is sufficiently accurate), then the director commits the time increment.

The Continuous director interoperates with all other timed Ptolemy II domains. Combining it with FSMs yields a particular form of modal model known as a hybrid system (Lee and Zheng, 2005; Lee, 2009). Combinations with discrete-event and synchronous/reactive domains are also useful (Lee and Zheng, 2007).

Ptera. The Ptera domain, described in Chapter 11, realizes a variant of event graphs. In Ptera, the components are not actors. Instead, the components are events, and the connections between components are triggering relations for events. A Ptera model represents how events in a system can trigger other events. Ptera is a timed model, and like FSM, it can be used to define the behavior of an actor to be used in another domain. In addition, events can be composites in that they have actions associated with them that are themselves defined by a submodel specified using another domain. Ptera is useful for specifying timed behaviors where input events may trigger chain reactions.

1.9 Case Study

Cyber-physical systems are intrinsically heterogeneous. CPS models, therefore, benefit from being able to combine models of computation. In this section, we walk through an example that uses several models of computation. The example is highly simplified, but with a little imagination, it is easy to see how the model can evolve to become an accurate and complete model of a large complex system. In particular, the large complex system we have in mind is an electric power system in a smart grid or on a vehicle (such as an airplane or advanced ground vehicle). In such a system, there are multiple sources of electric power (windmills, solar panels, turbines, backup generators, etc.) that must be coordinated to provide power to a multiplicity of loads. The system includes controllers that regulate the generators to keep voltages and frequencies near constant, and supervisory controllers that connect and disconnect loads and generators to provide services and handle faults to protect equipment. Such a system also includes networks whose dynamics may affect the overall behavior of the system.

Here, we illustrate a highly simplified version of such a system to show how the various MoCs come into play.

Example 1.4: A simplified model of a gas-powered generator that may be connected to and disconnected from a load is shown in Figure 1.8. This is a continuous-time model, as indicated by the Continuous director, which is discussed in Chapter 9. The model has two inputs, a *drive* signal, and a *loadAdmittance*. The output is a *voltage* signal. In addition, the model has three parameters, a time constant T, an output impedance Z, and a drive limit L. The model gives the output voltage of a generator over time as the generator gets more or less gas (specified by the *drive* input), and as the load varies (as specified by the *loadAdmittance* input).

This model exhibits simplified linear and nonlinear dynamics. The nonlinear dynamics is realized by the Limiter actor (see the sidebar on page 57), which limits

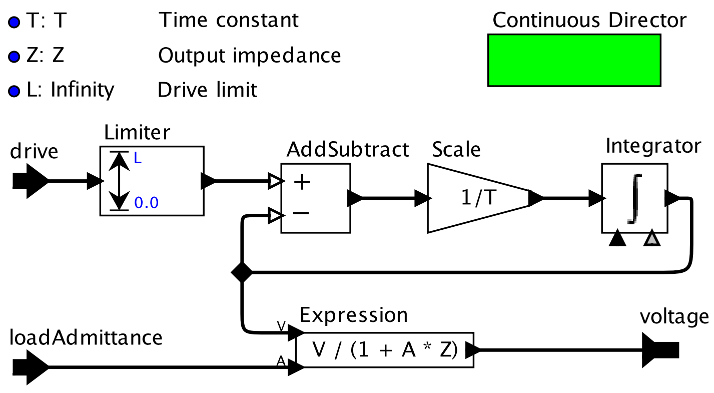


Figure 1.8: Simplified model of a gas-powered generator. [online]

the *drive* input. In particular, if the *drive* input becomes negative, it sets the drive to zero (you cannot extract gas from a generator). It also saturates the *drive* input at an upper bound given by the parameter L, which defaults to `Infinity`, meaning that there is no saturation (this generator can accept an arbitrarily large drive input).

The linear dynamics in this model is given by the small feedback loop, which includes an AddSubtract actor, a Scale actor, and an Integrator. If the output of the limiter is D, then this loop gives a value V that satisfies the following ordinary differential equation,

$$\frac{dV}{dt} = \frac{1}{T}(D - V) \, ,$$

where both D and V are functions of time (see Chapter 9 to understand how this model yields the above equation).

For our purposes here, understanding this equation is not important, since this part of such a model would typically be constructed by a mechanical engineer who is an expert in such models, but we can nevertheless make some intuitive observations. First, if $D = V$, then the derivative is zero, so the generator is stable and will produce an unchanging output. Second, when $D \neq V$, the feedback loop adjusts the value of V to make it closer to D. If $D > V$, then this equation makes the derivative of V positive, which means that V will increase. If $D < V$, then the derivative is negative, so V will decrease. In fact, the output V will converge to D exponentially with time constant T. A **time constant** is the amount of time that an exponential signal takes to reach $1 - 1/e \approx 63.2\%$ of its final (asymptotic) value.

The last part of the model is the part that models the effect of the load. This effect is modeled by the Expression actor (see Section 13.2.4), which uses Ohm's law to calculate the output voltage as a function of the value V (representing the generator's effort), the output impedance Z, and the load admittance A. An electrical engineer would recognize this calculation as the realization of a simple voltage divider.

For our purposes, it is sufficient to notice that if $A = 0$ (there is no load) or $Z = 0$ (the generator is an ideal voltage source with no output impedance), then the voltage output is equal to the effort V. A real generator, however, will have a non-zero output impedance. As the load admittance A increases from zero, the output voltage will drop.

The above model is about the simplest interesting model of **continuous dynamics**. To integrate this model with digital controllers, we could wrap the model in another one that defines the discrete interfaces, as shown next.

Example 1.5: The Generator model of Figure 1.8 is wrapped to provide a discrete interface in Figure 1.9. Here, the *drive* and *loadAdmittance* inputs go to instances of the ZeroOrderHold actor. These inputs, therefore, can be provided as discrete events rather continuous-time signals. The ZeroOrderHold actor converts these discrete events into continuous-time signals by holding the value constant between arrivals of events (see Section 9.2).

The output voltage goes through a PeriodicSampler actor (see Section 9.2), which produces discrete events that are samples of the output voltage. The sample period is a parameter *P* of the model.

This model exposes the time constant *T* and output impedance *Z*, but hides the drive limit *L*. Of course, the model designer could make other choices about which parameters to expose.

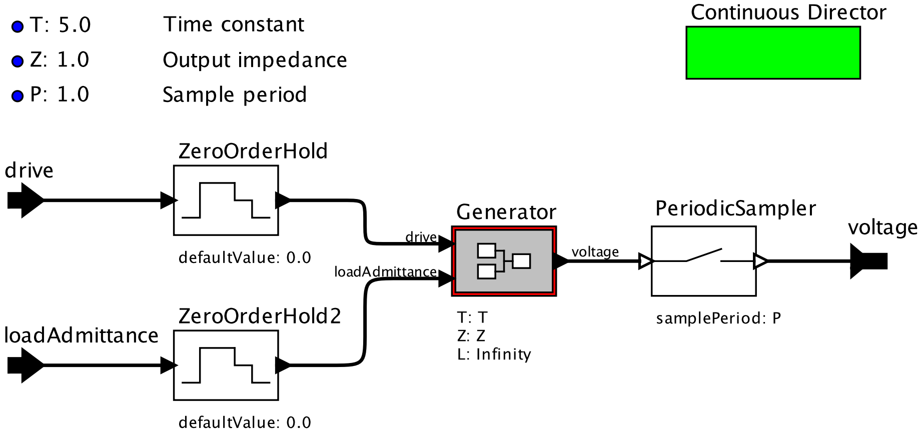


Figure 1.9: The Generator model of Figure 1.8 wrapped to provide a discrete interface. [online]

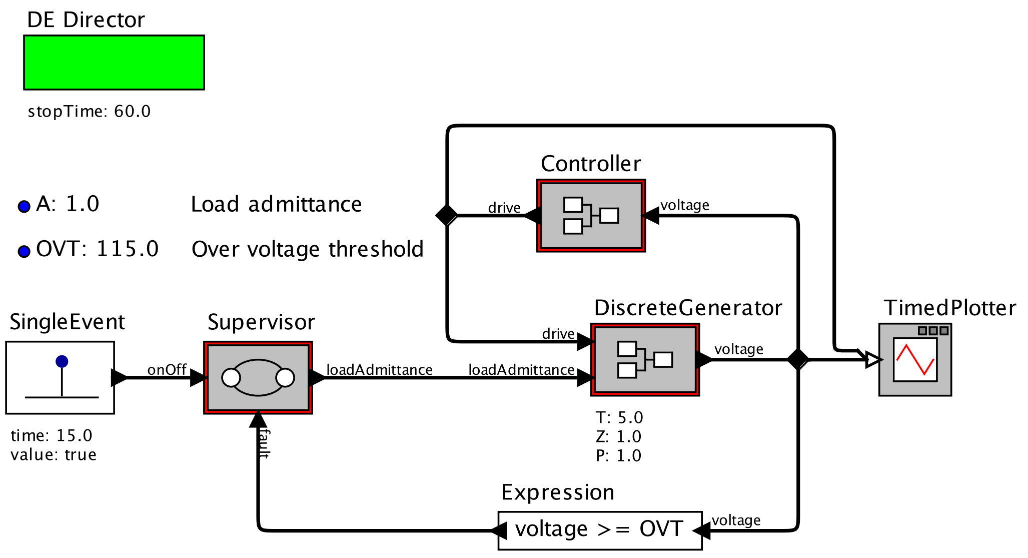


Figure 1.10: A discrete-event model with a generator, a controller, and an over-voltage protector. [online]

A continuous-time model may be embedded within a discrete-event model (see Chapter 7), as illustrated next.

Example 1.6: The DiscreteGenerator model of Figure 1.9 is embedded in a discrete-event model in Figure 1.10. This model has two parameters, the load admittance A and an over-voltage threshold OVT. The time constant T of the DiscreteGenerator is set to 5.0. This model includes two other components that we will explain below, a Supervisor, which provides the over-voltage protection, and a Controller, which regulates the *drive* input of the DiscreteGenerator based on measurements of the output voltage.

In addition, this model includes a simple test scenario, where a SingleEvent actor (see sidebar on page 241) requests that a load be connected at time 15.0, and a TimedPlotter actor (see Chapter 17), which displays the results of a run of the model, as shown in Figure 1.11.

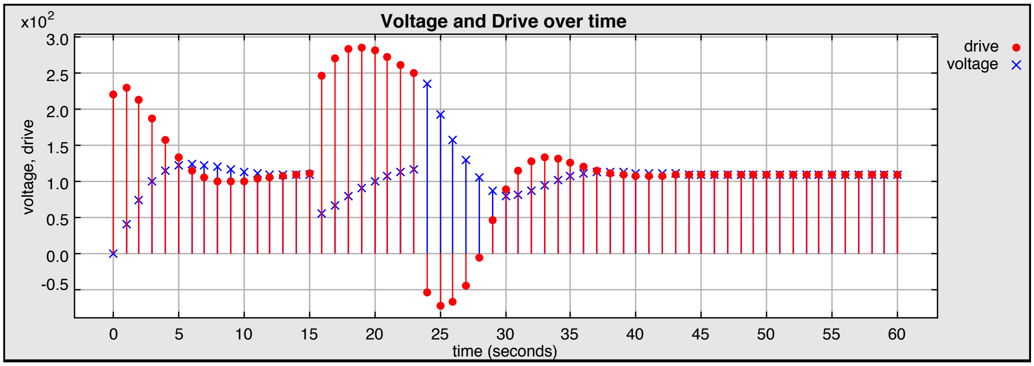


Figure 1.11: The plot produced by the model in Figure 1.10. [online]

In this test scenario, the load admittance is quite high (1.0) compared to the output impedance (also 1.0), so when the load is connected at time 15, the voltage abruptly drops to half its target value of 110 volts. The Controller compensates for this by substantially increasing the *drive*, but this causes the voltage to overshoot the target, and at time 24, to exceed the *OVT* threshold. The Supervisor reacts to this over-voltage condition by disconnecting the load, which causes the voltage to spike quite high, since the generator now has a substantial *drive* input. The Controller eventually brings the voltage back to the target level.

Notice further that when the load is disconnected, the Controller takes the *drive* signal negative. If this is a gas-powered generator, the Controller is trying to give the generator a negative flow of gas. Fortunately, our generator model includes a Limiter actor that prevents the model from actually providing that negative flow of gas.

The model in Figure 1.10 includes two very different kinds of controllers, a supervisory controller called Supervisor, and a low-level controller called simply Controller. These two controllers are specified using two additional MoCs, as explained next.

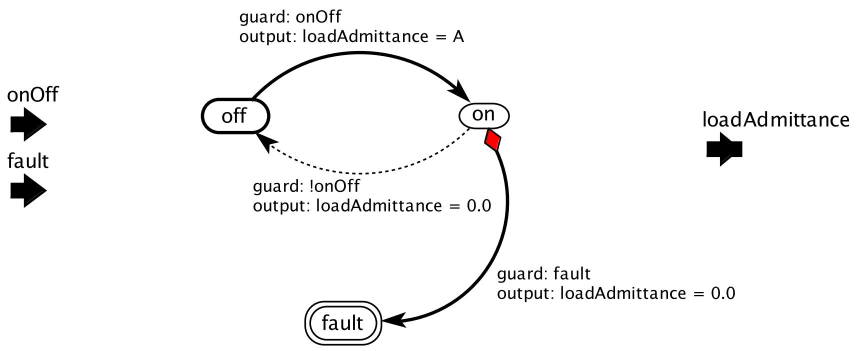

guard: onOff
output: loadAdmittance = A
onOff
off
on
loadAdmittance
fault
guard: !onOff
output: loadAdmittance = 0.0
guard: fault
output: loadAdmittance = 0.0
fault

Figure 1.12: The Supervisor of Figure 1.10. [online]

Example 1.7: The Supervisor model of Figure 1.10 is a finite state machine, shown in Figure 1.12. The notation here is explained in Chapter 6, but we can easily grasp the general behavior.

This FSM has two inputs, *onOff* (a boolean that requests to connect or disconnect the load) and *fault* (a boolean that indicates that an over-voltage condition has occurred). It has one output, *loadAdmittance*, which will be the actually load admittance provided to the generator.

The initial state of the FSM is *off*. When an *onOff* input arrives that has value true, the FSM will transition from the *off* state to the *on* state and produce a *loadAdmittance* output with value given by *A*, a parameter of the model. This connects the load.

When the FSM is in state *on*, if a *fault* event arrives with value true, then it will transition to the final state *fault* and set the *loadAdmittance* to 0.0, disconnecting the load. If instead an *onOff* event arrives with value false, then it will transition to the state *off* and also disconnect the load. The difference between these two transitions is that once the FSM has entered the *fault* state, it cannot reconnect the load without a system reset (which will bring the FSM back to the initial state).

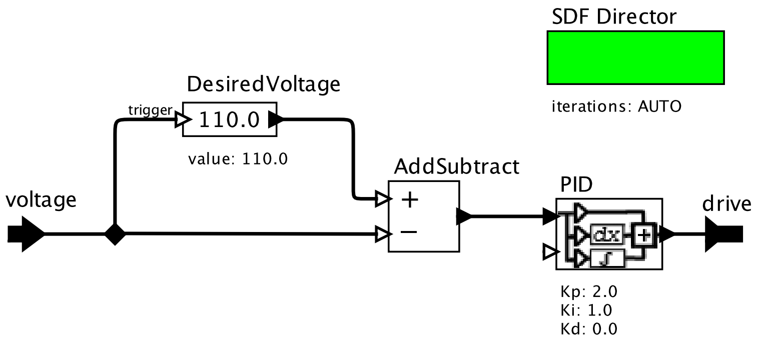


Figure 1.13: The Controller of Figure 1.10. [online]

Example 1.8: The Controller model of Figure 1.10 is the dataflow model shown in Figure 1.13. This model uses the SDF director (see Chapter 3), which is suitable for sampled-data signal processing. In this case, the controller compares the input *voltage* against a desired voltage (110 volts), and feeds the resulting error signal into a PID controller. A PID controller is a commonly used linear time-invariant system. A control engineer would know how to set the parameters of this controller, but in this case, we have simply chosen some parameters experimentally to yield an interesting test case.

Notice that the pieces of the model in Figure 1.10 are distinctly heterogeneous, touching on several disciplines within engineering and computer science. Typically, models of this type are the result of teams of engineers working together, and a framework that enables these teams to compose their models can become extremely valuable.

Many elaborations of this model are easy to envision. For example:

- The Generator could be defined as an actor-oriented class, so that it can be instantiated multiple times, and yet developed and maintained in a single centralized definition (see Section 2.6).
- The Generator model could be elaborated to reflect more sophisticated linear and non-linear dynamics using the techniques discussed in Chapter 9.

- The Generator model could be elaborated to include frequency and phase effects, for example by using complex-valued impedances and admittances together with a phasor representation.
- Models with a variable size (e.g., n generators and m loads, where n and m are parameters) could be created using the higher-order components considered in Section 2.7.
- The effects of network timing, clock synchronization, and contention for shared resources could be modeled using the techniques in Chapter 10.
- Signal processing techniques such as machine learning and spectral analysis, (see Chapter 3), could be integrated into the control algorithms.
- A units system could be included to make the model precise about the units used to measure time, voltage, frequency, etc.
- An ontology could be included to make the model precise about which signals and parameters represent voltages, admitances, impedances, etc., or even to make distinctions between domain-specific concepts such as the internal voltage (effort) of a generator vs. the voltage exhibited at its output, which is affected by its output impedance and load.

1.10 Summary

Ptolemy II focuses on actor-oriented modeling of complex systems, providing a disciplined approach to heterogeneity and concurrency. The central notion in 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 separation can dramatically increase the reusability of components and models. The remainder of this book shows how to build Ptolemy II models and how to leverage the properties of each of the models of computation.

2

Building Graphical Models

Christopher Brooks, Edward A. Lee, Stephen Neuendorffer, and John Reekie

Contents

This chapter provides a tutorial on constructing Ptolemy II simulation models using **Vergil**, the Ptolemy graphical user interface (**GUI**). Figure 2.1 shows a simple Ptolemy II model in Vergil. This model is shown in the graph editor, one of several possible entry mechanisms available in Ptolemy II. It is also possible, for example, to define models in Java or XML.

2.1 Getting Started

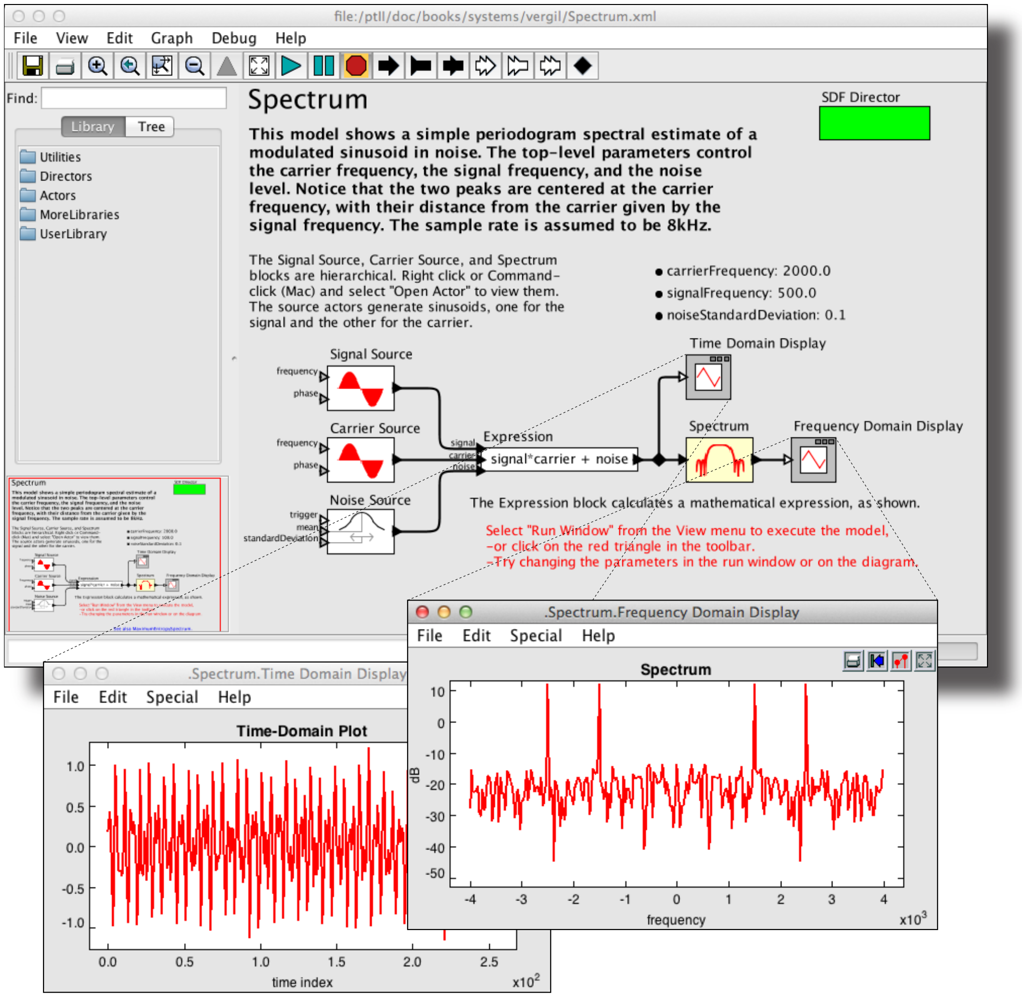
Executing the examples in this chapter requires installation of Ptolemy II[1]. Once it is installed, you will need to invoke Vergil, which will display the initial welcome window shown in Figure 2.2. The "Tour of Ptolemy II" link takes you to the page shown in Figure 2.3.

2.1.1 Executing a Pre-Built Signal Processing Example

On the "Tour of Ptolemy II" page, the first example listed under "Basic Modeling Capabilities" (Spectrum), is the model shown in Figure 2.1. This model creates a sinusoidal signal, multiplies it by a sinusoidal carrier, adds noise, and then estimates the power spectrum. This model can be executed using the run button in the toolbar (the blue triangle

[1]See `http://ptolemy.org/ptolemyII/ptIIlatest` for the latest release, and `http://chess.eecs.berkeley.edu/ptexternal/` for access to the ongoing development version. Alternatively, most of the figures in this book have an online version of the model that you can browse using any web browser. More interestingly, if you are reading this book on a Java-capable machine, you can also follow a link that uses Java **Web Start** to launch Vergil without any explicit installation step. You can browse, edit, and execute models, and also save them to local disk. For information on Web Start, see `http://en.wikipedia.org/wiki/Java_Web_Start`.

pointing to the right). Two signal plots will then be displayed in their own windows, as shown in Figure 2.1. The plot on the right shows the power spectrum and the plot on the left shows the time-domain signal. Note the four peaks, which indicate the modulated sinusoid. You can adjust the frequencies of the signal and the carrier as well as the amount



Spectrum

This model shows a simple periodogram spectral estimate of a modulated sinusoid in noise. The top-level parameters control the carrier frequency, the signal frequency, and the noise level. Notice that the two peaks are centered at the carrier frequency, with their distance from the carrier given by the signal frequency. The sample rate is assumed to be 8kHz.

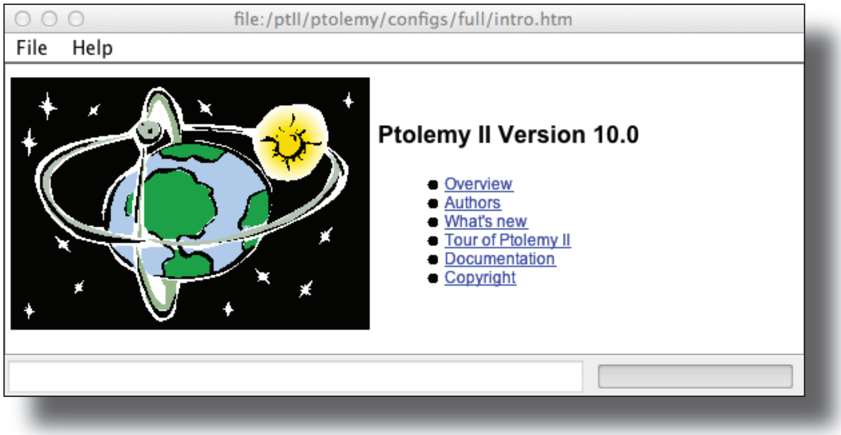
The Signal Source, Carrier Source, and Spectrum blocks are hierarchical. Right click or Command-click (Mac) and select "Open Actor" to view them. The source actors generate sinusoids, one for the signal and the other for the carrier.

- carrierFrequency: 2000.0
- signalFrequency: 500.0
- noiseStandardDeviation: 0.1

The Expression block calculates a mathematical expression, as shown.

Select "Run Window" from the View menu to execute the model,
-or click on the red triangle in the toolbar.
-Try changing the parameters in the run window or on the diagram.

Figure 2.1: Example of a Vergil window and the windows that result from running the model. [online]



Ptolemy II Version 10.0

- Overview
- Authors
- What's new
- Tour of Ptolemy II
- Documentation
- Copyright

Figure 2.2: Initial welcome window.

of noise by double clicking on those parameters in the block diagram in Figure 2.1 near the upper right of the window.

The icon (the graphical block) for the Expression actor displays the expression that will be calculated:

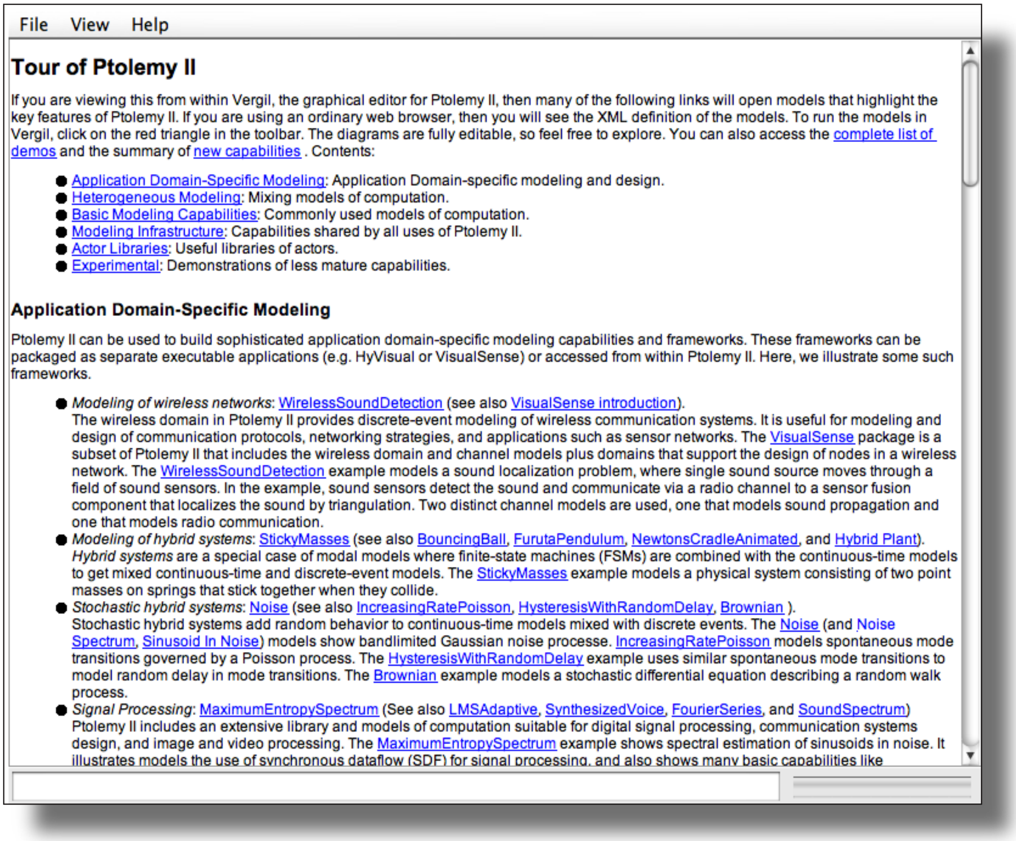
```
signal*carrier + noise
```

The identifiers in this expression, `signal`, `carrier`, and `noise`, refer to the input ports. The Expression actor is flexible; it can have any number of input ports (which can have arbitrary names) and it uses a rich expression language to specify the value of the output as a function of the inputs. It can also specify parameters for the model in which the expression is contained. (The expression language is described in Chapter 13.)

Right clicking and selecting [`Documentation→Get Documentation`] displays documentation for that actor. Figure 2.4 shows the documentation for the Expression actor.

Three of the actors in Figure 2.1 are composite actors, meaning that their implementation is itself a Ptolemy II model. You can invoke the `Open Actor` context menu[2] to reveal the

[2] A **context menu** is a menu that is specific to the object under the cursor. It is obtained by right clicking the mouse (or control clicking, if the mouse does not have a right button) while the cursor is over the icon.



Tour of Ptolemy II

If you are viewing this from within Vergil, the graphical editor for Ptolemy II, then many of the following links will open models that highlight the key features of Ptolemy II. If you are using an ordinary web browser, then you will see the XML definition of the models. To run the models in Vergil, click on the red triangle in the toolbar. The diagrams are fully editable, so feel free to explore. You can also access the complete list of demos and the summary of new capabilities . Contents:

- Application Domain-Specific Modeling: Application Domain-specific modeling and design.
- Heterogeneous Modeling: Mixing models of computation.
- Basic Modeling Capabilities: Commonly used models of computation.
- Modeling Infrastructure: Capabilities shared by all uses of Ptolemy II.
- Actor Libraries: Useful libraries of actors.
- Experimental: Demonstrations of less mature capabilities.

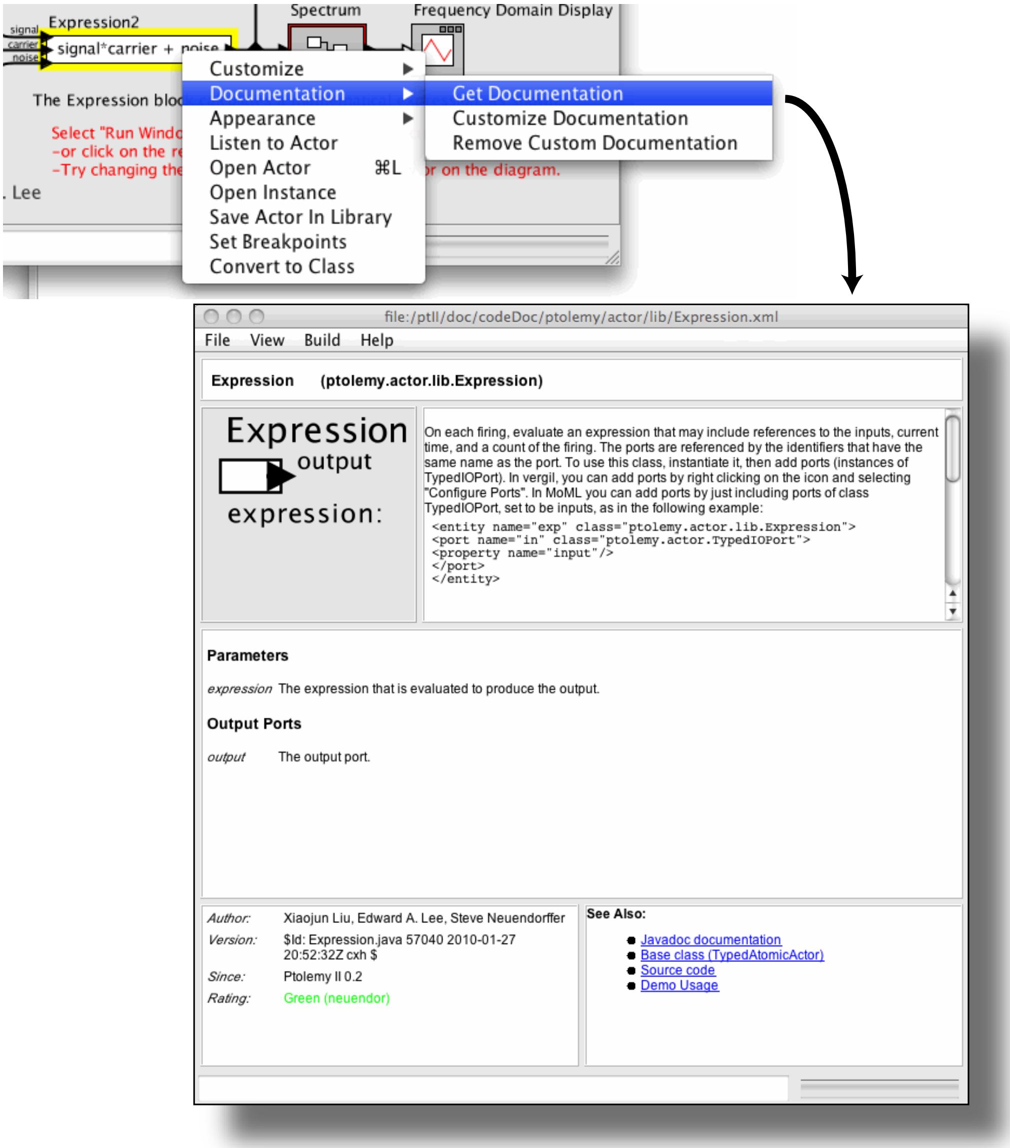
Application Domain-Specific Modeling

Ptolemy II can be used to build sophisticated application domain-specific modeling capabilities and frameworks. These frameworks can be packaged as separate executable applications (e.g. HyVisual or VisualSense) or accessed from within Ptolemy II. Here, we illustrate some such frameworks.

- *Modeling of wireless networks*: WirelessSoundDetection (see also VisualSense introduction).
The wireless domain in Ptolemy II provides discrete-event modeling of wireless communication systems. It is useful for modeling and design of communication protocols, networking strategies, and applications such as sensor networks. The VisualSense package is a subset of Ptolemy II that includes the wireless domain and channel models plus domains that support the design of nodes in a wireless network. The WirelessSoundDetection example models a sound localization problem, where single sound source moves through a field of sound sensors. In the example, sound sensors detect the sound and communicate via a radio channel to a sensor fusion component that localizes the sound by triangulation. Two distinct channel models are used, one that models sound propagation and one that models radio communication.
- *Modeling of hybrid systems*: StickyMasses (see also BouncingBall, FurutaPendulum, NewtonsCradleAnimated, and Hybrid Plant).
Hybrid systems are a special case of modal models where finite-state machines (FSMs) are combined with the continuous-time models to get mixed continuous-time and discrete-event models. The StickyMasses example models a physical system consisting of two point masses on springs that stick together when they collide.
- *Stochastic hybrid systems*: Noise (see also IncreasingRatePoisson, HysteresisWithRandomDelay, Brownian).
Stochastic hybrid systems add random behavior to continuous-time models mixed with discrete events. The Noise (and Noise Spectrum, Sinusoid In Noise) models show bandlimited Gaussian noise processe. IncreasingRatePoisson models spontaneous mode transitions governed by a Poisson process. The HysteresisWithRandomDelay example uses similar spontaneous mode transitions to model random delay in mode transitions. The Brownian example models a stochastic differential equation describing a random walk process.
- *Signal Processing*: MaximumEntropySpectrum (See also LMSAdaptive, SynthesizedVoice, FourierSeries, and SoundSpectrum)
Ptolemy II includes an extensive library and models of computation suitable for digital signal processing, communication systems design, and image and video processing. The MaximumEntropySpectrum example shows spectral estimation of sinusoids in noise. It illustrates models the use of synchronous dataflow (SDF) for signal processing, and also shows many basic capabilities like

Figure 2.3: The tour of Ptolemy II page.

Signal Source implementation, as shown in Figure 2.5. This block diagram shows how the sinusoidal signal is generated.



Expression **(ptolemy.actor.lib.Expression)**

On each firing, evaluate an expression that may include references to the inputs, current time, and a count of the firing. The ports are referenced by the identifiers that have the same name as the port. To use this class, instantiate it, then add ports (instances of TypedIOPort). In vergil, you can add ports by right clicking on the icon and selecting "Configure Ports". In MoML you can add ports by just including ports of class TypedIOPort, set to be inputs, as in the following example:

```
<entity name="exp" class="ptolemy.actor.lib.Expression">
<port name="in" class="ptolemy.actor.TypedIOPort">
<property name="input"/>
</port>
</entity>
```

Parameters

expression The expression that is evaluated to produce the output.

Output Ports

output The output port.

Author:	Xiaojun Liu, Edward A. Lee, Steve Neuendorffer
Version:	$Id: Expression.java 57040 2010-01-27 20:52:32Z cxh $
Since:	Ptolemy II 0.2
Rating:	Green (neuendor)

See Also:

- Javadoc documentation
- Base class (TypedAtomicActor)
- Source code
- Demo Usage

Figure 2.4: Viewing documentation for actors.

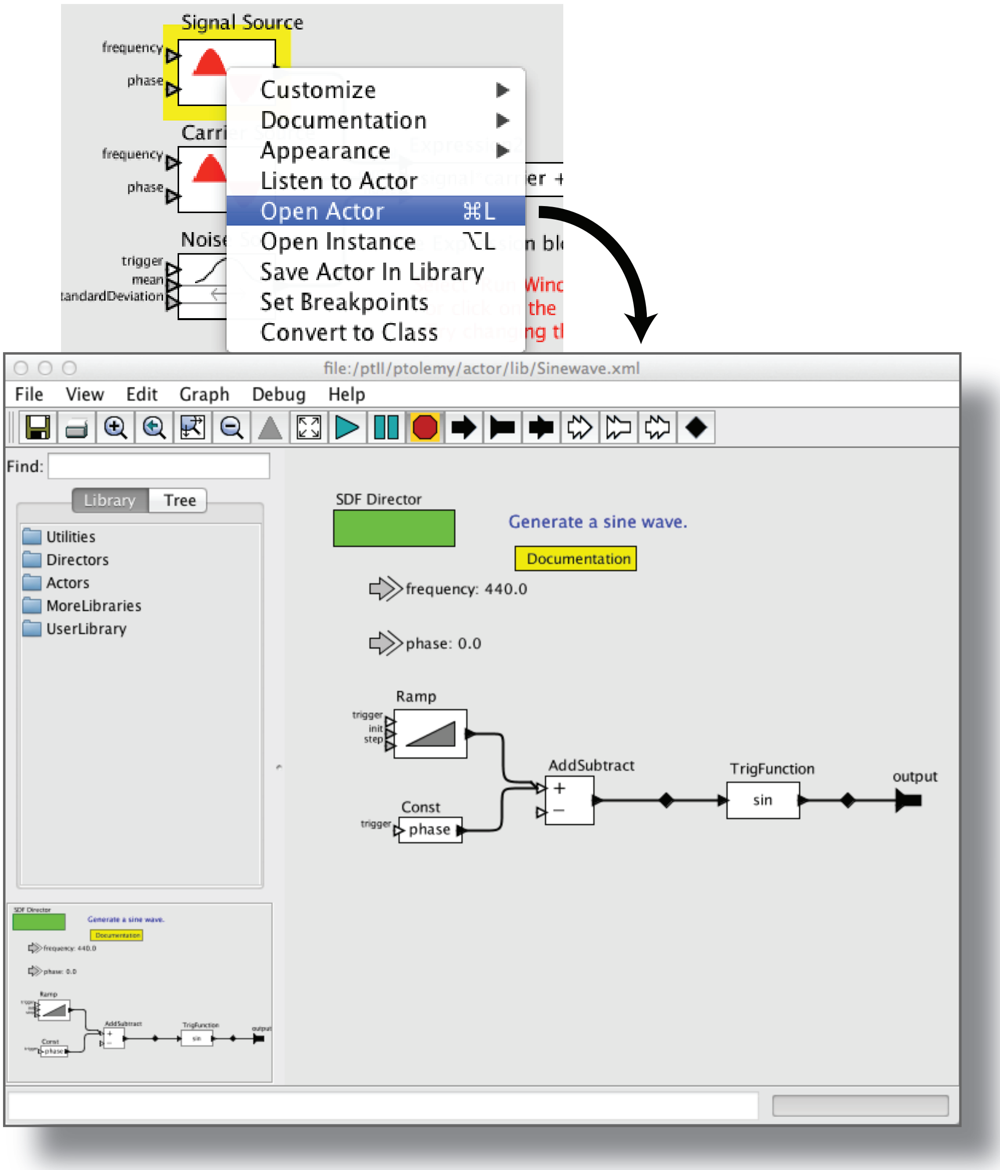


Figure 2.5: Invoke `Open Actor` on composite actors to reveal their implementation.

2.1.2 Creating and Running a Model

Create a new model by selecting [File→New→Graph Editor] in the menu bar. You should see something like the window shown in Figure 2.6. The left-hand side of the page shows a library of components that can be dragged into the model-building area. Perform the steps outlined below to create a simple model.

- Open the Actors library and then Sources. Find the Const actor under Sources→Generic and drag an instance into the model-building area.
- Open Sinks→GenericSinks and drag a Display actor onto the page (see boxes on pages 48 and 49).

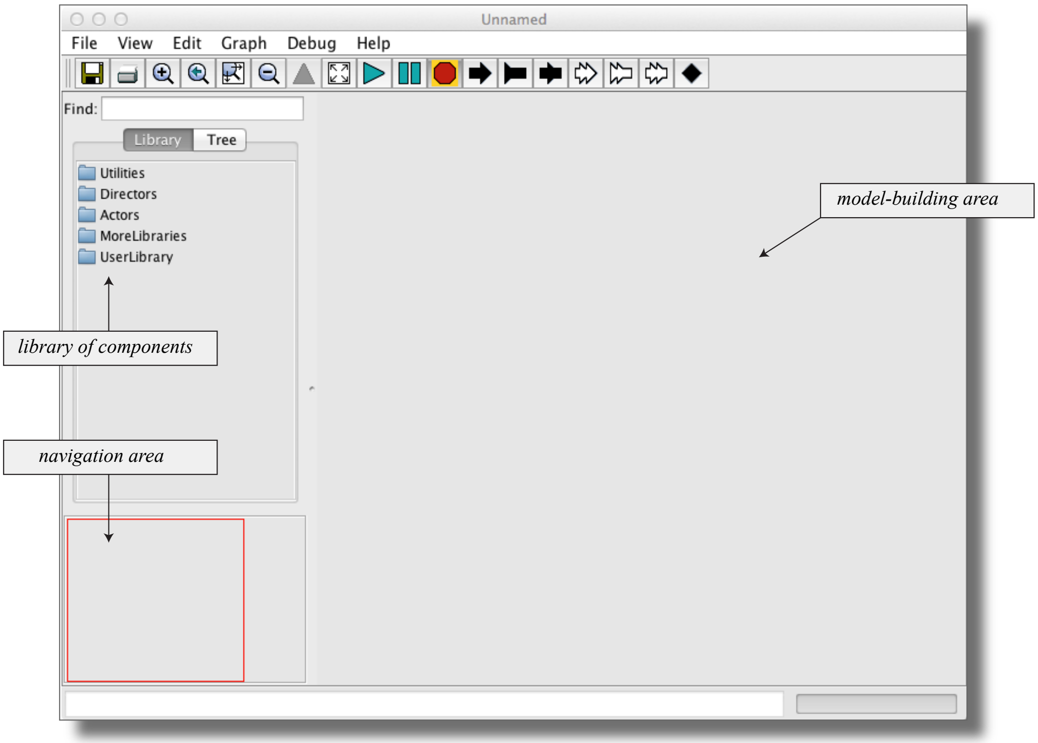


Figure 2.6: An empty Vergil Graph Editor.

- Drag a connection from the output port on the right of the Const actor to the input port of the Display actor.
- Open the `Directors` library and drag the SDF Director onto the page. The director controls various aspects of the model's functionality, such as (for example) how many iterations the model will execute (which is, by default, just one iteration).

Now you should have something that looks like Figure 2.7. In this model, the Const actor will create a string and the Display actor will display that string. Set the string variable to "Hello World" by double or right clicking on the Const actor icon and selecting [`Customize→Configure`], which will display the dialog box shown in Figure 2.8. En-

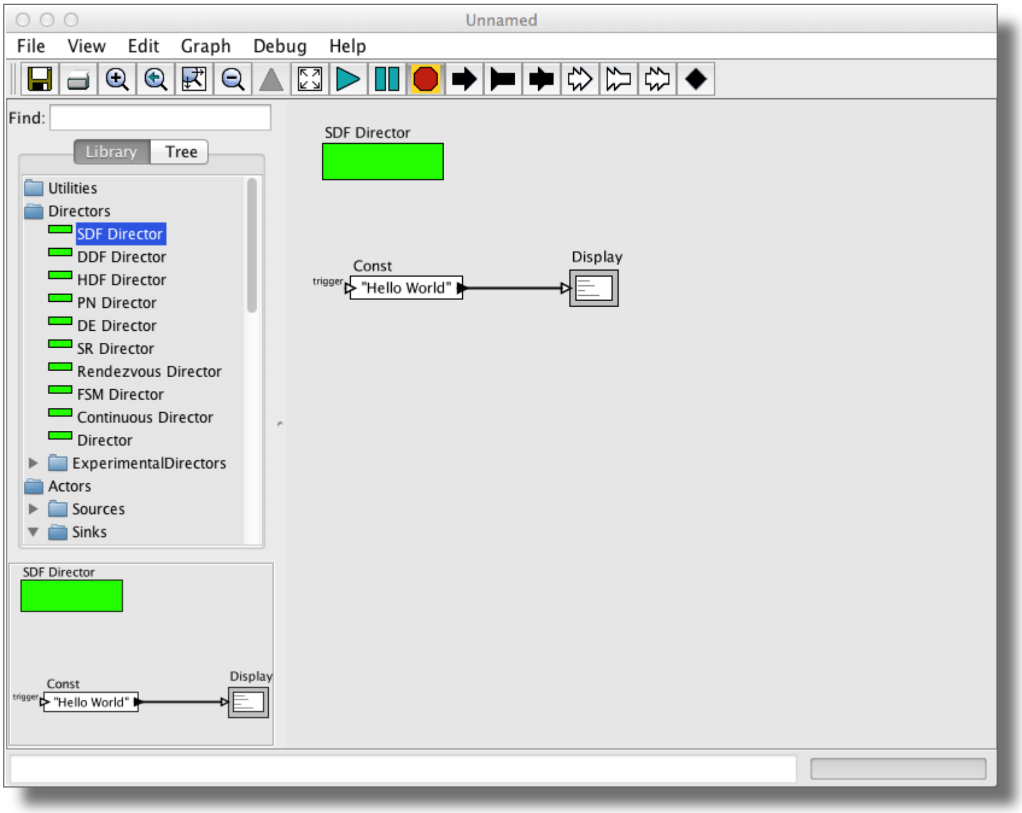
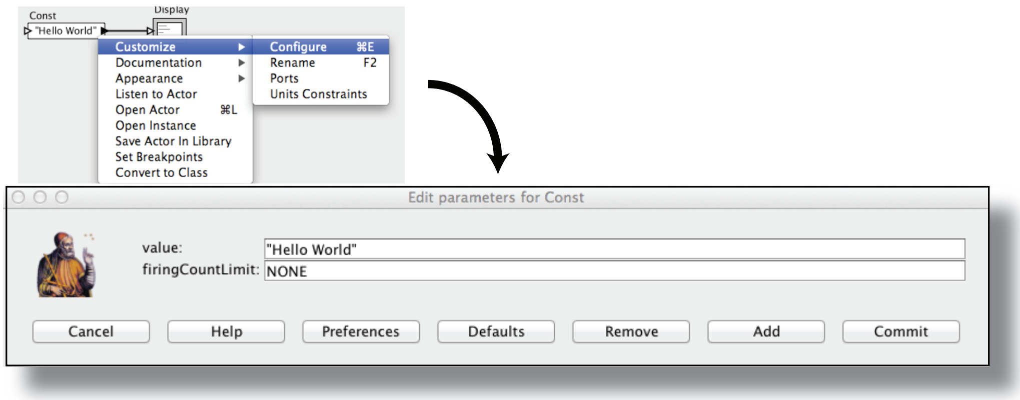


Figure 2.7: The Hello World example. [online]



value: "Hello World"
firingCountLimit: NONE

Figure 2.8: The Const parameter editor.

ter the string `"Hello World"` (with the quotation marks) for the *value* parameter and click the Commit button. (The quotation marks ensure that the expression will be interpreted as a string.) If you run this model, a display window will show the text "Hello World."

You may wish to save your model using the File menu. File names for Ptolemy II models should end in ".xml" or ".moml" so that Vergil will properly process the file the next time it is opened.

2.1.3 Making Connections

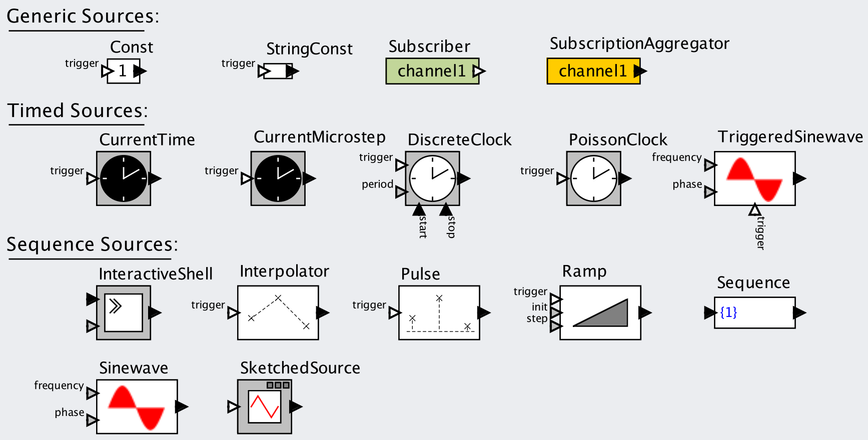
The model constructed above contains two actors and one connection between them. If you move either actor (by clicking and dragging), the connection will be re-routed automatically. We can now explore how to create and manipulate more complicated connections.

First, create a model in a new graph editor that includes an SDF Director, a Ramp actor (found in the `Sources→SequenceSources` library), a Display actor, and a SequencePlotter actor (found in the `Sinks→SequenceSinks` library), as shown in Figure 2.9.

Suppose we wish to route the output of the Ramp actor to both the Display and the SequencePlotter. In this case, we need an explicit relation in the diagram. A **relation** is a

Sidebar: Sources Library

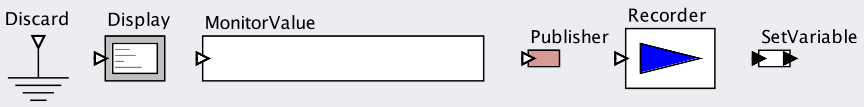
The three `Actors→Sources` libraries contain sources of signals.



- **Const** produces a value (or expression) set by a parameter (see Chapter 13).
- **StringConst** produces strings and may reference other parameters (see Section 13.2.3).
- **Subscriber** outputs data received from a Publisher (described in the Sinks Library sidebar on page 49).
- **SubscriptionAggregator** combines data from multiple Publishers using a specified operation (currently addition or multiplication).
- **CurrentTime**, **CurrentMicrostep**, **DiscreteClock**, and **PoissonClock** are timed sources described in Chapter 7.
- **TriggeredSinewave** and **Sinewave** produce sinusoidal outputs based on either the current time or a specific sample rate, respectively.
- **InteractiveShell** outputs the value entered in a shell window opened during execution.
- **Interpolator** and **Pulse** both generate waveforms, one by interpolating between values, and the other by zero-filling between values.
- **Ramp** produces a steadily increasing or decreasing output sequence.
- **Sequence** produces an arbitrary sequence of values, possibly periodic.
- **SketchedSource** produces a sequence specified interactively with the mouse.

Sidebar: Sinks Library

Actors in the `Actors→Sinks` library serve as destinations for signals. Most are plotters, described in Chapter 17. The few that are not are contained in the `Sinks→GenericSinks` library, and shown below. All of these sinks accept any data type at their inputs.



- **Discard** discards all inputs. In most domains, leaving an output disconnected has the same effect (the Rendezvous domain is an exception).
- **Display** displays the values of inputs in a text window that opens when the model is executed.
- **MonitorValue** displays its inputs in the Vergil icon that displays the model.
- **Publisher** establishes named connections to instances of Subscriber and SubscriptionAggregator (described in the Sources sidebar on page 48).
- **Recorder** stores the values of inputs internally; custom Java code is then needed to access those values.
- **SetVariable** sets the value of a variable or parameter defined in the model. Since this variable may be read by another actor that has no connection to the SetVariable actor, SetVariable may introduce nondeterminism into a model. That is, the results of executing the model may depend on arbitrary scheduling decisions that determine whether SetVariable executes before actors that read the affected variable. To mitigate this risk, SetVariable has a parameter called *delayed* that by default is set to true. When it is true, the affected variable will only be set at the end of the current iteration of the director. For most directors (but notably, not for PN or Rendezvous), this setting will ensure deterministic behavior. The SetVariable actor has an output port that can be used to ensure that other specified actors are executed only after SetVariable executes. This approach can also eliminate nondeterminacy, even if *delayed* is set to false.

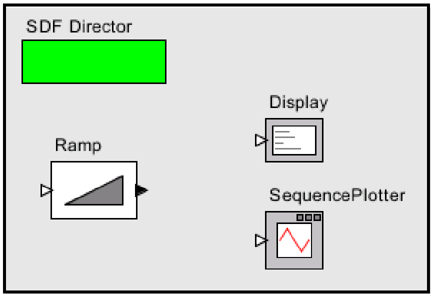


Figure 2.9: Three unconnected actors in a model.

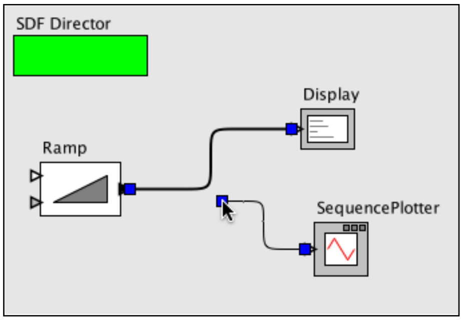


Figure 2.10: Dragging a connection to an existing one will create a three-way connection by creating an explicit relation.

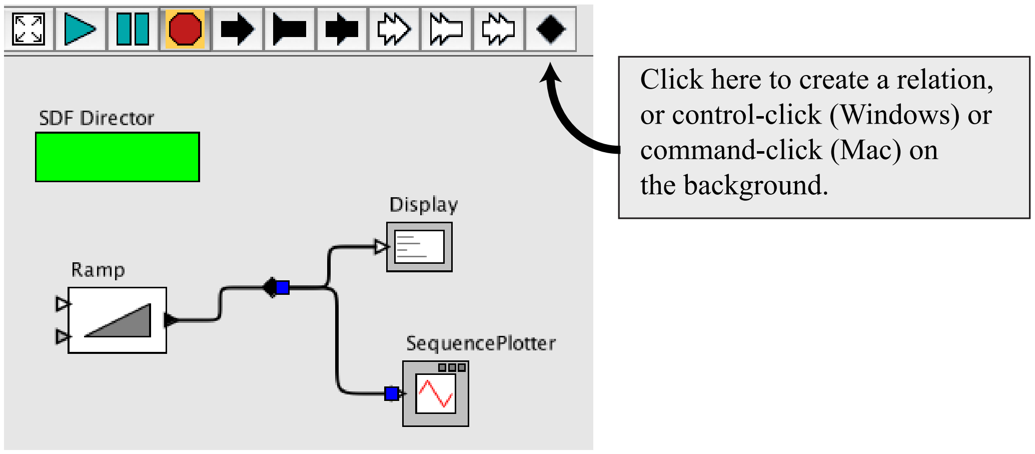

SDF Director
Display
Ramp
SequencePlotter
Click here to create a relation, or control-click (Windows) or command-click (Mac) on the background.

Figure 2.11: A relation can also be created by clicking on the black diamond in the toolbar.

splitter that enables connecting more than two ports; it is represented in the diagram by a black diamond, as shown in Figure 2.11. It can be created in several ways:

- Drag the endpoint of a link into the middle of an existing line, as shown in Figure 2.10.
- Control-click[3] on the background. A relation will appear.
- Click on the button in the toolbar with the black diamond on it, as shown in Figure 2.11.

Note that if you simply click and drag on the relation, the relation is selected and moved, which does not result in a connection. To make a connection, hold the control key while clicking and dragging on the relation.

In the model shown in Figure 2.11, the relation is used to broadcast the output from a single port to a number of places. The single port still has only one connection — a connection to the relation.

Even with the simple diagrams we have created so far, it can become tedious to arrange the icons and control the routing of wires between ports and relations. Fortunately, Ptolemy II includes a sophisticated automated layout tool, which you can find in the [`Graph`→

[3]On a Macintosh, use command-click. (This substitution applies throughout the book.)

`Automatic Layout`] menu command (or invoke using Control-T). This tool was contributed by Spönemann et al. (2009). Occasionally, however, you may still need to manually adjust the layout. To specifically control the routing of connections, you can use multiple relations along a connection and independently control the position of each one. Multiple relations used in a single connection are called a **relation group**. In a relation group, there is no significance to the order in which relations are linked.

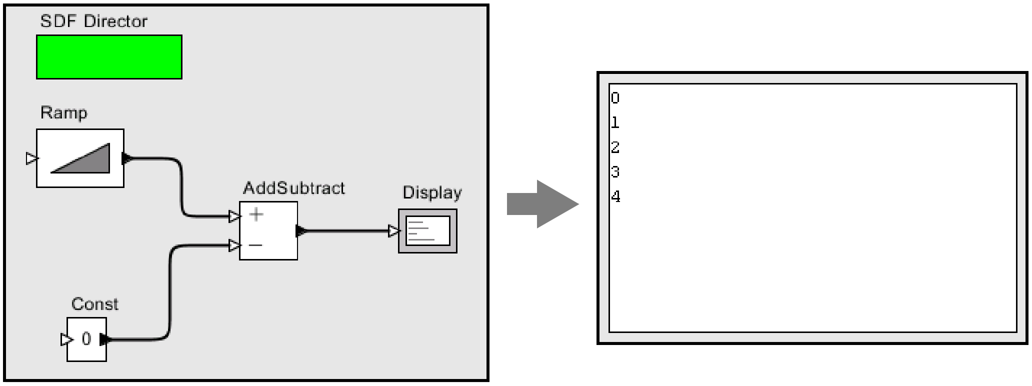
Ptolemy II supports two types of ports, which are indicated by filled and unfilled triangles. Filled triangles designate single-input (or single-output) ports, while unfilled triangles allow multiple inputs (or outputs). For example, the output port of the Ramp actor is a **single port**, while the input port of the Display and SequencePlotter actors are **multiports**. In multiports, each connection is treated as a separate **channel**.

Choose another signal source from the library and add it to the model. Connect this additional source to the SequencePlotter or to the Display, thus implementing a multiport input to those blocks. Not all data type inputs will be accepted, however; the SequencePlotter, for example, can only accept inputs of type *double* or types that can be losslessly converted to *double*, such as *int*. In the next section, we discuss data types and their use in Ptolemy II models.

2.2 Tokens and Data Types

In the example of Figure 2.7, the Const actor creates a sequence of values on its output port. Each value in the sequence is called a **token**. A token is created by one actor, sent through an output port, and received by one or more destination actors, each of which retrieves the token through an input port. In this case, the Display actor will receive the tokens produced by the Const actor and display them in a window. A token is simply a unit of data, such as a string or numerical value, that is communicated between two actors via ports.

The tokens produced by the Const actor can have any value that can be expressed in the Ptolemy II expression language (see Chapter 13). Try setting the value to `1` (the integer with value one), or `1.0` (the floating-point number with value one), or `{1.0}` (a single-entry array containing 1.0), or `{value=1,name="one"}` (a record with two elements: an integer named "value" and a string named "name"), or even `[1,0;0,1]` (the two-by-two identity matrix). These are all valid expressions that can be used in the Const actor.



```
0
1
2
3
4
```

Figure 2.12: Another example, used to explore data types. [online]

The Const actor is able to produce data with different **types**, and the Display actor is able to display data with different types. Most actors in the actor library are **polymorphic actors**, meaning that they can operate on or produce data with multiple types, though their specific processing behavior may be different for different types. Multiplying matrices, for example, is not the same operation as multiplying integers, but both can be accomplished using the MultiplyDivide actor in the `Math` library. Ptolemy II includes a sophisticated type system that allows actors to process different data types efficiently and safely. The type system was created by Xiong (2002) and is described in Chapter 14.

To explore data types further, create the model in Figure 2.12, using the SDF Director. The Ramp actor is listed under the `Sources→SequenceSources` sublibrary, and the AddSubtract actor is listed under the `Math` library (see box on page 57). Set the *value* parameter of the Const to be 0 and the *iterations* parameter of the director to 5. Running the model will result in the display of five consecutive numbers starting at 0 and ending at 4, as shown in the figure. These values are produced by subtracting the constant output of the Const actor from the current value of the Ramp. Experiment with changing the value of the Const actor and see how it changes the five numbers at the output.

Now change the value of the Const actor back to `"Hello World"`. When you execute the model, you should see an **exception** window, as shown in Figure 2.13. This error is caused by attempting to subtract a string value from an integer value. This error is one kind of **type error**.

The actor that caused the exception is highlighted, and the name of the actor is shown in the exception. In Ptolemy II models, all objects have a dotted name. The dots separate elements in the hierarchy. Thus, ".helloWorldSubtractError.AddSubtract" is an object named "AddSubtract" contained by a object named ".helloWorldAddSubtractError". (The model was saved as a file called `helloWorldAddSubtract.xml`.)

Exceptions can be a very useful debugging tool, particularly when developing components in Java. To illustrate how to use them, click on the Display Stack Trace button in the exception window of Figure 2.13. You should see a stack trace like that shown in Figure 2.13. This window displays the execution sequence that resulted in the exception. For example, if you scroll towards the bottom, text similar to the following will be displayed:

```
at ptolemy.data.StringToken._subtract(StringToken.java:359)
```

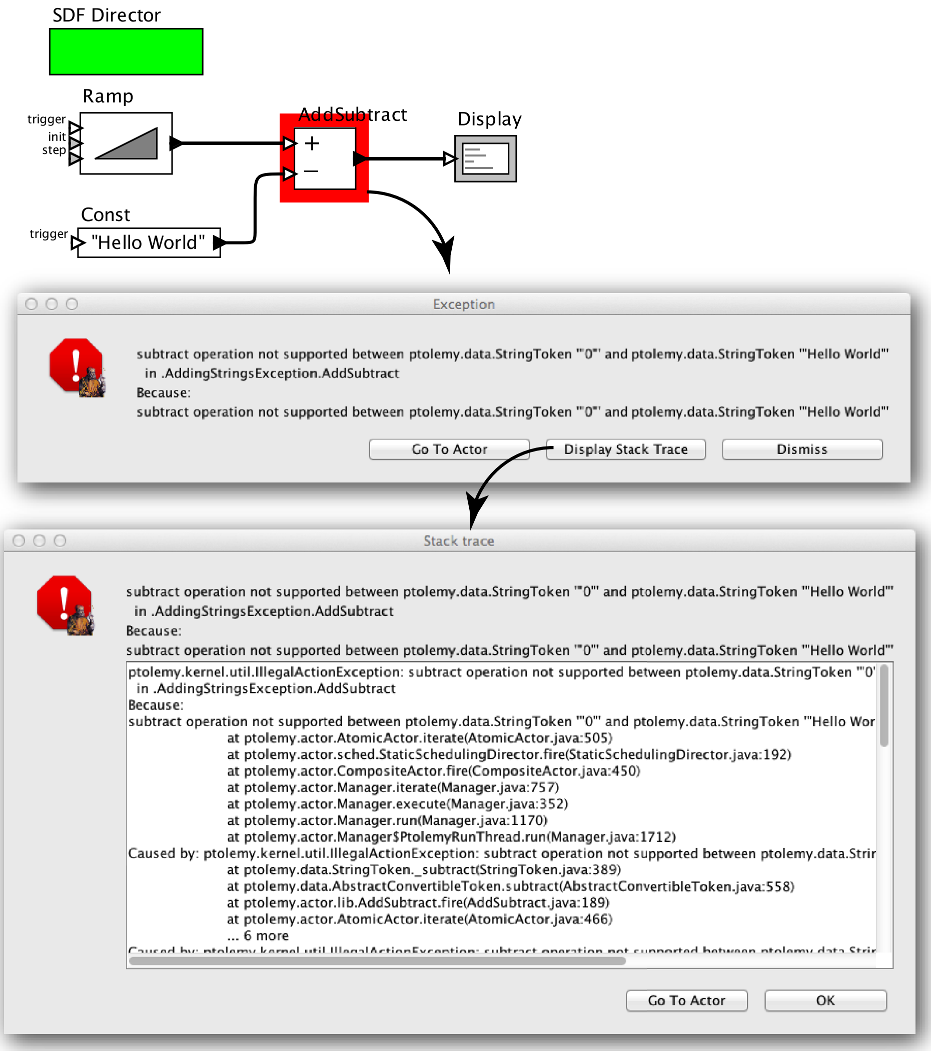
This line indicates that the exception occurred within the `subtract` method of the class ptolemy.data.StringToken, at line 359 of the source file StringToken.java. Since Ptolemy II is distributed with source code (and most installation mechanisms support installation of the source), this can be very useful information. For type errors, you probably do not need to see the stack trace, but if you have extended the system with your own Java code or you encounter a subtle error that you do not understand, then looking at the stack trace can be very illuminating.

To find the file StringToken.java referred to above, find the Ptolemy II installation directory. If that directory is `$PTII`, then the location of this file is given by the full class name, but with the periods replaced by slashes; in this case, it is at

```
$PTII/ptolemy/data/StringToken.java
```

The slashes will be backslashes under Windows.

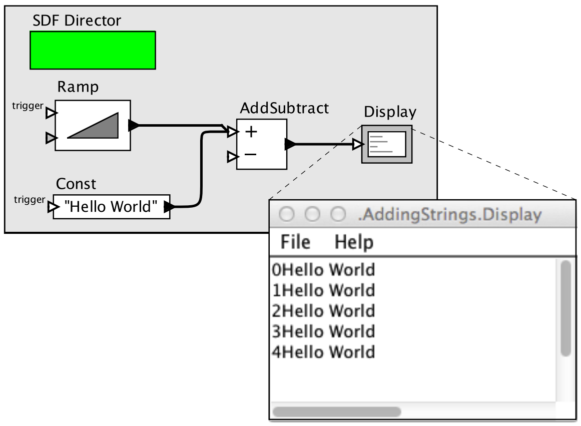
Let's try a small change to the model to eliminate the exception. Disconnect the Const actor from the lower port of the AddSubtract actor and connect it instead to the upper port, as shown in Figure 2.14. You can do this by selecting the connection, deleting it (using the delete key), and adding a new connection–or by selecting it and dragging one of its endpoints to the new location. Notice that the upper port is an unfilled triangle; this indicates that you can make more than one connection to it. Now when you run the model you should see a sequence of string values starting with "`0Hello World`", as shown in



```
subtract operation not supported between ptolemy.data.StringToken "0" and ptolemy.data.StringToken "'Hello World'"
  in .AddingStringsException.AddSubtract
Because:
subtract operation not supported between ptolemy.data.StringToken "0" and ptolemy.data.StringToken "'Hello World'"
```

```
subtract operation not supported between ptolemy.data.StringToken "0" and ptolemy.data.StringToken "'Hello World'"
  in .AddingStringsException.AddSubtract
Because:
subtract operation not supported between ptolemy.data.StringToken "0" and ptolemy.data.StringToken "'Hello World'"
ptolemy.kernel.util.IllegalActionException: subtract operation not supported between ptolemy.data.StringToken "0'
  in .AddingStringsException.AddSubtract
Because:
subtract operation not supported between ptolemy.data.StringToken "0" and ptolemy.data.StringToken "'Hello Wor
            at ptolemy.actor.AtomicActor.iterate(AtomicActor.java:505)
            at ptolemy.actor.sched.StaticSchedulingDirector.fire(StaticSchedulingDirector.java:192)
            at ptolemy.actor.CompositeActor.fire(CompositeActor.java:450)
            at ptolemy.actor.Manager.iterate(Manager.java:757)
            at ptolemy.actor.Manager.execute(Manager.java:352)
            at ptolemy.actor.Manager.run(Manager.java:1170)
            at ptolemy.actor.Manager$PtolemyRunThread.run(Manager.java:1712)
Caused by: ptolemy.kernel.util.IllegalActionException: subtract operation not supported between ptolemy.data.Strir
            at ptolemy.data.StringToken._subtract(StringToken.java:389)
            at ptolemy.data.AbstractConvertibleToken.subtract(AbstractConvertibleToken.java:558)
            at ptolemy.actor.lib.AddSubtract.fire(AddSubtract.java:189)
            at ptolemy.actor.AtomicActor.iterate(AtomicActor.java:466)
            ... 6 more
```

Figure 2.13: An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from integers. Examining the stack trace for such exceptions can sometimes be useful for debugging your model, particularly if it includes custom actors. [online]



```
0Hello World
1Hello World
2Hello World
3Hello World
4Hello World
```

Figure 2.14: Addition of a string to an integer. [online]

the figure. Following Java conventions, strings can be added (using concatenation), but they cannot be subtracted.

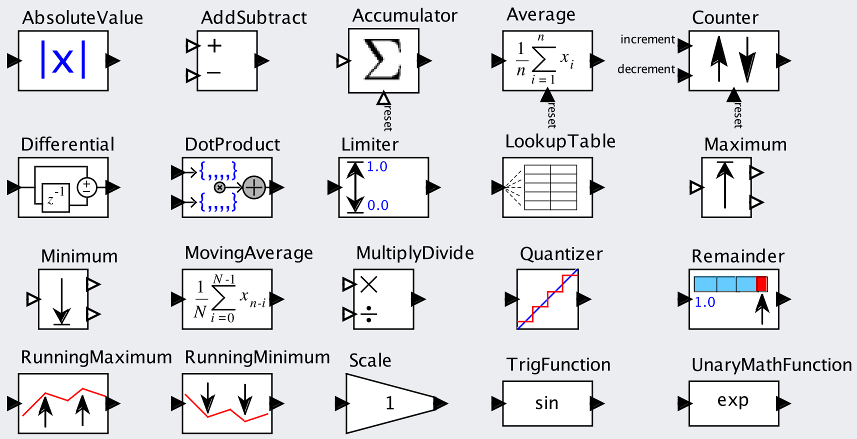
All the connections to a multiport must have the same type. In this case, the multiport has a sequence of integers coming in (from the Ramp) and a sequence of strings (from the Const). The Ramp integers are automatically converted to strings and concatenated with "Hello World" to generate the output sequence.

As illustrated by the above example, Ptolemy II automatically performs type conversions when required by the actor, if possible. As a rough guideline, Ptolemy II will perform automatic type conversions when there is no loss of information. An *int* can be converted to a *string*, for example, but not vice versa. An *int* can be converted to a *double*, but not vice versa. An *int* can be converted to a *long*, but not vice versa. The details are explained in Chapter 14, but it is usually not necessary to remember the conversion rules. Typically, the data type conversions and resulting computations will perform as would be expected.

To further explore data types, try modifying the Ramp so that its parameters have different types. For example, try making *init* and *step* strings.

Sidebar: Math Library

Actors in the `Actors→Math` library perform mathematical operations, and are shown below (except the most versatile, Expression, explained in Chapter 13).



These are mostly self explanatory:

- **AbsoluteValue** computes the absolute value of the input.
- **AddSubtract** adds tokens at the *plus* inputs, and subtracts tokens at the *minus* inputs.
- **Accumulator** outputs the sum of all tokens that have arrived.
- **Average** outputs the average of all tokens that have arrived.
- **Counter** counts up or down when inputs are received on the two input ports.
- **Differential** outputs the difference between the current input and the previous one.
- **DotProduct** computes the inner product of two array or matrix inputs.
- **Limiter** limits the value of the input to a specified range of values.
- **LookupTable** performs a lookup into a table provided as an array.

(Continued on page 58.)

Sidebar: Math Library (Continued)

- **Maximum** outputs the maximum of the currently available input tokens.
- **Minimum** outputs the minimum of the currently available input tokens.
- **MovingAverage** outputs the average of some number of the most recent inputs.
- **MultiplyDivide** multiplies the tokens on the *multiply* inputs, and divides by the tokens on the *divide* inputs.
- **Quantizer** outputs the nearest value to the input from a specified list of values.
- **Remainder** outputs the remainder after dividing the input by the *divisor* parameter.
- **RunningMaximum** outputs the maximum value seen so far at the input.
- **RunningMinimum** outputs the minimum value seen so far at the input.
- **Scale** multiplies the input by a constant given as a parameter.
- **TrigFunction** performs trigonometric functions, including cosine, sine, tangent, arccosine, arcsine, and arctangent.
- **UnaryMathFunction** performs various math functions with a single argument, including exponentiation, logarithm, sign, squaring, and square root.

There are subtleties. Some actors with multiple input ports (AddSubtract, Counter, and MultiplyDivide) or with multiport inputs (Maximum and Minimum), do not require input tokens on all input channels. When these actors fire, they operate on whatever input tokens are available. For example, if AddSubtract has no tokens on any *plus* input channel, and only one token on a *minus* input channel, then the output will be the negative of that one token. Whether inputs are available when the actor fires depends on the director and the model. With the SDF director, exactly one input token is provided on every input channel for every firing.

These actors are **polymorphic**; they can operate on a variety of data types. For example, the AbsoluteValue actor accepts inputs of any scalar type (see Chapter 14). If the input type is *complex*, it computes the magnitude. For other scalar types, it computes the absolute value.

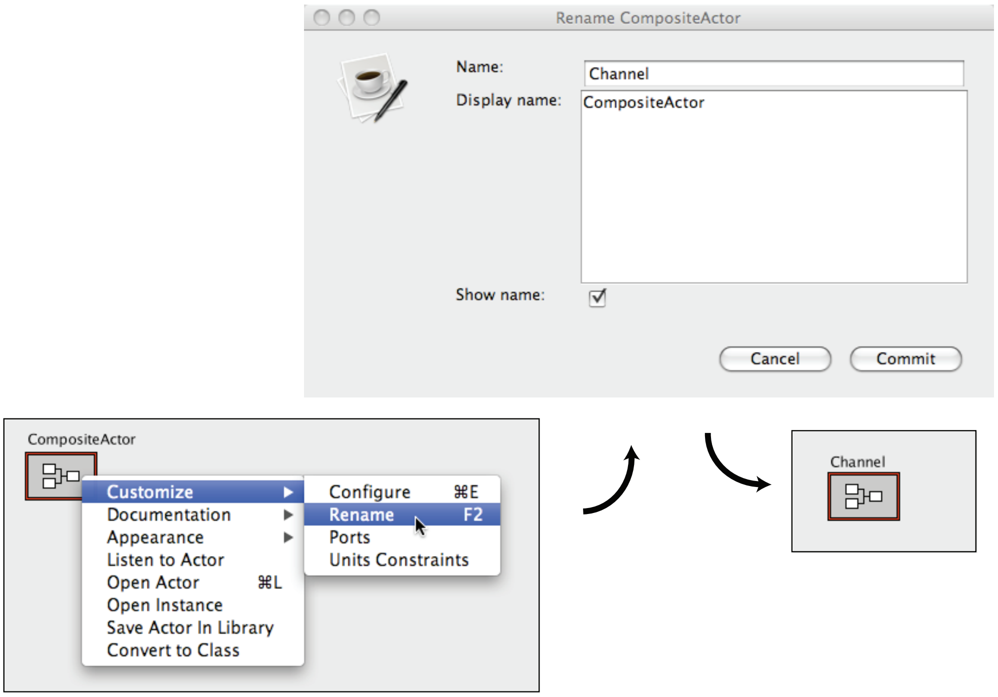
Accumulator and Average both have *reset* input ports (at the bottom of the icon). They calculate the sum or average of sequences of inputs, and they can be reset.

2.3 Hierarchy and Composite Actors

Ptolemy II supports (and encourages) hierarchical models. These are models that contain components that are themselves models. These components are called composite actors.

Consider the typical signal processing problem of recovering a signal from a noisy channel. Using Ptolemy II, we will create a composite actor modeling a communication channel that adds noise, and then use that actor within a larger model.

To create the composite actor, drag a **CompositeActor** from the `Utilities` library. In the context menu (obtained by right clicking over the composite actor), select [`Customize →Rename`], and give the composite an appropriate name, like **Channel**, as shown in Figure 2.15. (Note that you can also supply a *Display name*, which is arbitrary text that will be displayed instead of the name of the actor.) Then, using the context menu again, select


Rename CompositeActor

Name: Channel

Display name: CompositeActor

Show name: ✓

Figure 2.15: Changing the name of an actor.

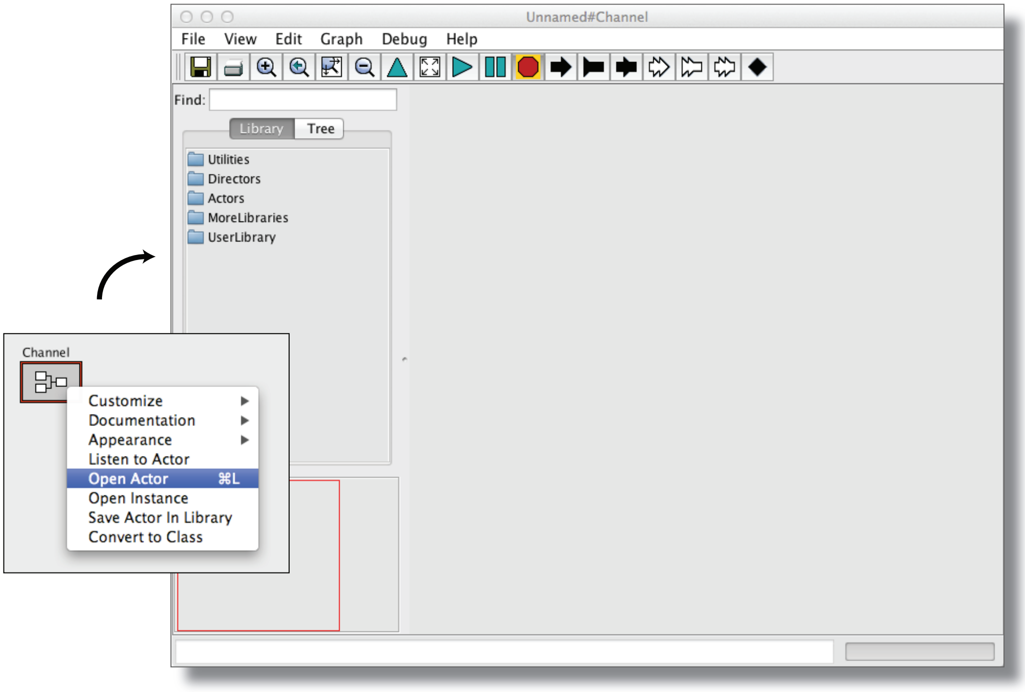


Figure 2.16: Opening a new composite actor, which shows the blank inner actor.

`Open Actor`. This will open a new window with an empty graph editor, as shown in Figure 2.16. Note that the original graph editor is still open; to see it, move the new graph editor window aside by dragging the title bar of the window. Alternatively, you can click on the upward pointing triangle in the toolbar to navigate back to the container.

2.3.1 Adding Ports to a Composite Actor

The newly created composite actor needs input and output ports. There are several ways to add them, the easiest of which is to click on the port buttons in the toolbar. The port buttons are shown as white and black arrowheads on the toolbar, as described in Figure 2.17. You can explore the ports in the toolbar by hovering the mouse over each button; a tool tip will pop up that describes the button.

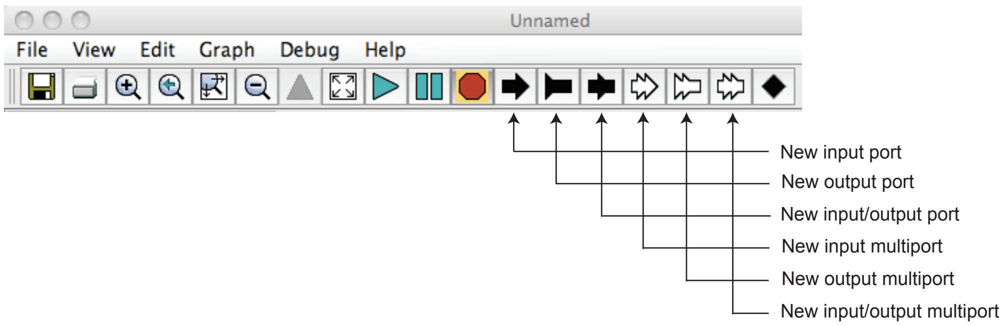
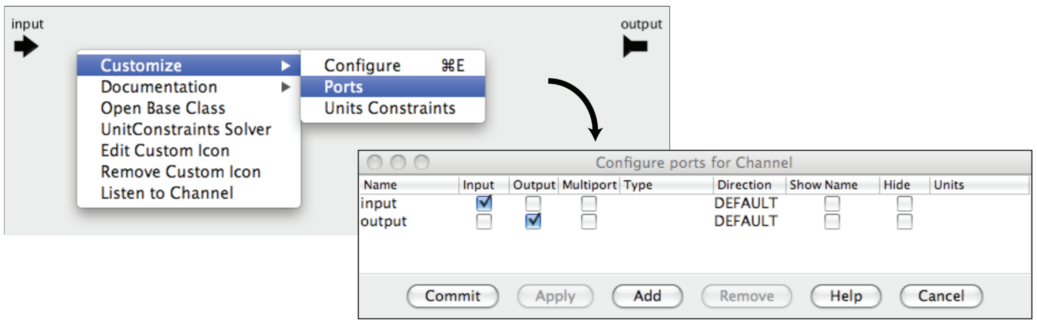


Figure 2.17: Summary of toolbar buttons for creating new ports.

Create an input port and an output port and rename them *input* and *output* by right clicking on the ports and selecting [Customize→Rename]. Note that, as shown in Figure 2.18, you can also right click on the background of the composite actor and select [Customize →Ports] to add ports, remove ports, or change whether a port is an input, an output, or a multiport. The resulting dialog box also allows you to set the type of the port, though you will typically not need to set the port type since Ptolemy II determines the type from the



Name	Input	Output	Multiport	Type	Direction	Show Name	Hide	Units
input	✓	☐	☐		DEFAULT	☐	☐	
output	☐	✓	☐		DEFAULT	☐	☐	

Figure 2.18: Right clicking on the background brings up a dialog that can be used to configure ports.

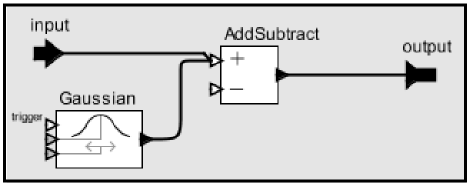


Figure 2.19: A simple channel model defined as a composite actor.

port's connections. You can also specify the direction of a port[4] and whether the name of the port is shown outside the icon (by default it is not), or even whether the port is shown at all.

Using these ports, create the diagram shown in Figure 2.19[5]. The **Gaussian** actor, which is in the `Random` library, creates values from a Gaussian distributed random variable. If you then navigate back to the container, you should be able to easily create the model shown in Figure 2.20. The Sinewave actor is listed under `Sources→SequenceSources`, and the SequencePlotter actor under `Sinks→SequenceSinks`. The Sinewave actor is also a composite actor (try opening the actor). If you execute this model (you will probably want to set the *iterations* parameter of the director to something reasonable, like 100), you should see a plot similar to the one in Figure 2.20.

2.3.2 Setting the Types of Ports

In the above example, it was not necessary to define the port types. Their types were inferred from the connections and constants in the model, as explained in Chapter 14. Occasionally, however, you will need to set the types of the ports. Notice in Figure 2.18 that there is a column in the dialog box that enables the port type to be specified. To specify that a port has type boolean, for example, you could enter *boolean*. This will only have an effect, however, if the port is contained by an opaque composite actor (one that has its own director). In the model you have built, the Channel composite actor is transparent, and setting the types of its ports will have no effect. Transparent composite

[4]The direction of a port is defined by where it appears on the icon for the actor. By default, input ports appear on the left, output ports on the right, and ports that are both inputs and outputs appear on the bottom of the icon.

[5]Hint: to create a connection starting on one of the external ports, hold down the control key when dragging, or on a Macintosh, the command key.

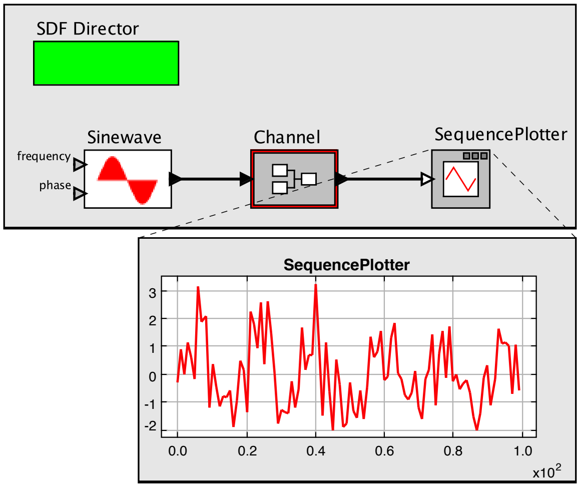


Figure 2.20: A simple signal processing example that adds noise to a sinusoidal signal. [online]

actors are merely a notational convenience, and the ports of the composite play no role in the execution of the model.

Commonly used types include *complex*, *double*, *fixedpoint*, *float*, *general*, *int*, *long*, *matrix*, *object*, *scalar*, *short*, *string*, *unknown*, *unsignedByte*, *xmlToken*, *arrayType(int)*, *arrayType(int, 5)*, *[double]* and {*x=double, y=double*}. A detailed description of types appears in Chapter 14.

In the Ptolemy II expression language, square braces are used for matrices, and curly braces are used for arrays. An array is an ordered list of tokens of arbitrary type. A Ptolemy II matrix is a one- or two-dimensional structure containing numeric types. To specify a double matrix port type, for example, you would use the following expression:

```
[double]
```

This expression creates a 1 x 1 matrix containing a *double* (the value of which is irrelevant here). It serves as a prototype to specify a double matrix type. Similarly, we can specify an array of complex numbers as

```
{complex}
```

A record has an arbitrary number of data elements, each of which has a name and a value, where the value can be of any type. To specify a record containing a string called "name" and an integer called "address" you would use the following expression to specify the type:

```
{name=string, address=int}
```

Details about arrays, matrices, and records are given in Chapter 13.

2.3.3 Multiports, Busses, and Hierarchy

As explained above in Section 2.1.3, a multiport handles multiple independent channels. In a similar manner, a relation can handle multiple independent channels. A relation has a *width* parameter, which by default is set to Auto, meaning that the width is inferred from the context. If the width is not unity, it will be displayed as shown in Figure 2.21

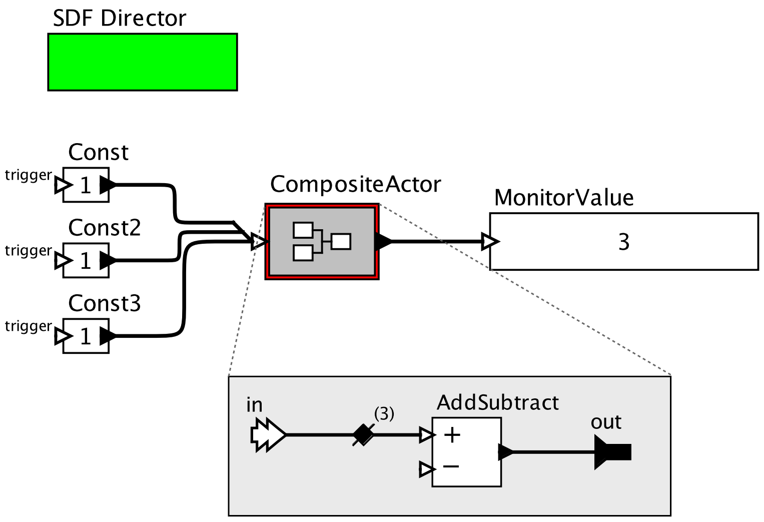


Figure 2.21: Relations can have a width greater than one, meaning that they carry more than one channel. The width of the relation inside the composite in this figure is inferred to be three. [online]

as a number adjacent to a slash through a relation. Such a connection is called a **bus**, because it carries multiple signals. If in the example in Figure 2.21 you set the width of the relation inside the composite to 2, then only two of the three channels will be used inside the composite.

In most circumstances, the width of a relation is inferred from the usage (Rodiers and Lickly, 2010), but occasionally it is useful to set it explicitly. You can also explicitly construct a bus using the **BusAssembler** actor, or split one apart using a **BusDisassembler** actor, both found in the `FlowControl→Aggregators` library.

2.4 Annotations and Parameterization

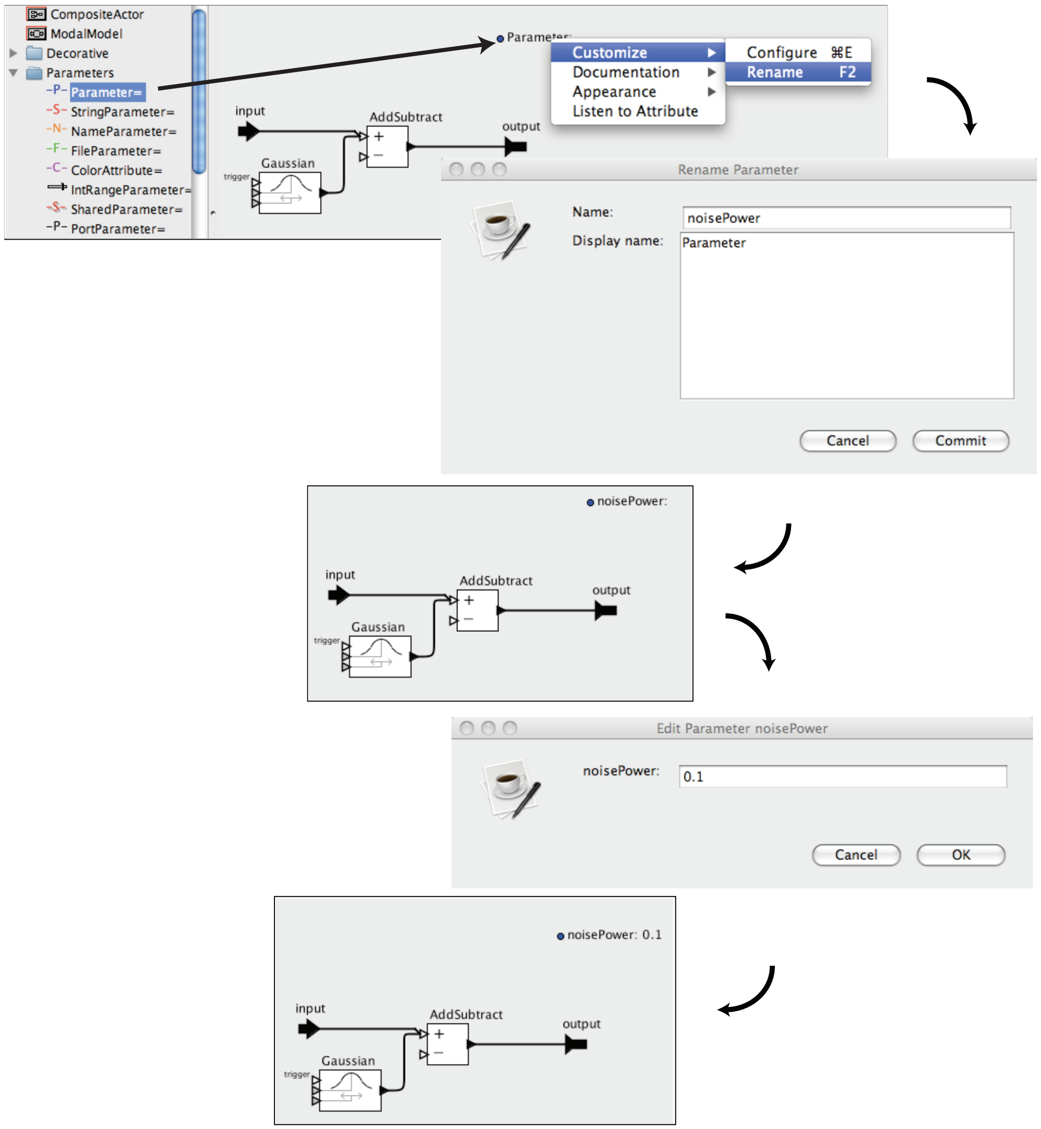
In this section, we will enhance the model in Figure 2.20 in several ways. We will add parameters, insert decorative and documentary annotations, and customize the actor icons.

2.4.1 Parameters in Hierarchical Models

First, notice from Figure 2.20 that the noise overwhelms the sinusoid, making it barely visible. A useful modification to this channel model would be to add a **parameter** that sets the level of the noise. To make this change, open the channel model by right clicking on it and selecting `Open Actor`. In the channel model, add a parameter by dragging one in from the sublibrary `Utilities→Parameters`, as shown in Figure 2.22. Right click on the parameter to rename it to `noisePower`. (To use a parameter in expressions, its name cannot have spaces.) Right click (or double click) on the parameter to change its default value to 0.1.

This parameter can now be used to set the amount of noise. The Gaussian actor has a parameter called *standardDeviation*. Since the noise power is equal to the variance, not the standard deviation, change the *standardDeviation* parameter so that its value is `sqrt(noisePower)`, as shown in Figure 2.23. (See Chapter 13 for details on the expression language.)

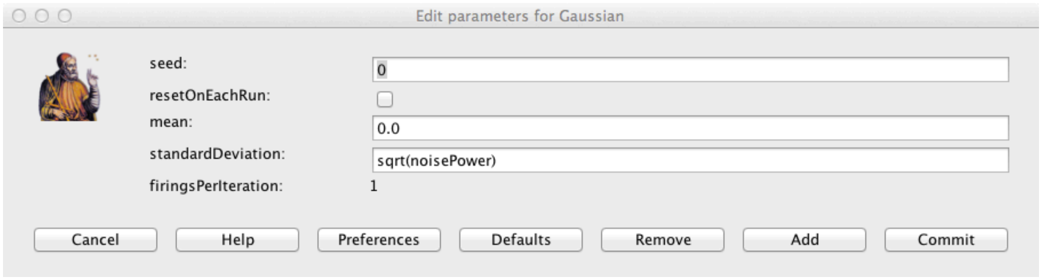
To observe the effect of the parameter, return to the top-level model and edit the parameters of the Channel actor (by either double clicking or right clicking and selecting [`Customize→Configure`]). Change the *noisePower* from the default 0.1 to 0.01. Run the model. You should now get a relatively clean sinusoid like that shown in Figure 2.24.



Name: noisePower
Display name: Parameter

noisePower: 0.1

Figure 2.22: Adding a parameter to the Channel model. [online]



seed: 0

resetOnEachRun: ☐

mean: 0.0

standardDeviation: sqrt(noisePower)

firingsPerIteration: 1

Figure 2.23: The standard deviation of the Gaussian actor is set to the square root of the noise power.

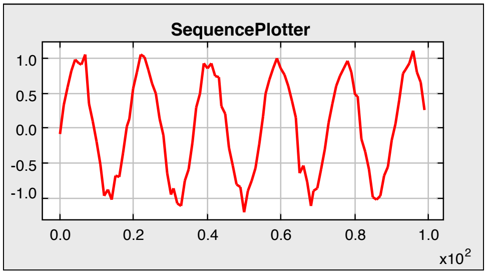


Figure 2.24: The output of the simple signal processing model in Figure 2.20 with noise power = 0.01. [online]

You can also add parameters to a composite actor by clicking on the "Add" button in the edit parameters dialog for the Channel composite. This dialog is accessed by double clicking on the Channel icon, by right clicking and selecting [`Customize→Configure`], or by right clicking on the background inside the composite and selecting [`Customize→Configure`]. A key point to note here is that parameters that are added this way will not be visible in the diagram, so this mechanism should be used sparingly and only when there is a good reason to hide parameters from someone browsing the model.

Finally, note that it is possible to create an object called a **port parameter** or **parameter port** that is both a parameter and a port. The *frequency* and *phase* objects in Figure 2.5 are port parameters. They can be accessed in an expression like any other parameter, but when an input arrives at the parameter port during execution, the value of the parameter gets updated. To create a port parameter object in a model, simply drag one in from the `Utilities→Parameters` library and assign it a name.

2.4.2 Decorative Elements

The model can also be enhanced with a variety of decorative elements — that is, elements that affect its appearance but not its functionality. Such elements can improve the readability and aesthetics of a model. For example, try dragging an *Annotation* from the `Utilities→Decorative` sublibrary, and creating a title for the diagram. Such annotations are highly recommended; they correspond to comments in programs and can greatly improve readability. Other decorative elements (such as geometric shapes) can be dragged into the diagram from the same library.

2.4.3 Creating Custom Icons

Vergil provides an icon editor to enable users to create custom actor icons. To create a custom icon, right click on the standard icon and select [`Appearance→Edit Custom Icon`], as shown in Figure 2.25. The box in the middle of the icon editor displays the size of the default icon, for reference. Try creating an icon like the one shown in Figure 2.26. Hint: The fill color of the rectangle is set to `none` and the fill color of the trapezoid is first set using the color selector, then modified to have an *alpha* (transparency) of 0.5. Finally, since the icon itself has the actor name in it, the [`Customize→Rename`] dialog is used to deselect *show name*.

2.5 Navigating Larger Models

Some models are too large to view on one screen. There are four toolbar buttons, shown in Figure 2.27, that permit zooming in and out. The “Zoom reset” button restores the zoom factor to its original value, and “Zoom fit” calculates the zoom factor so that the entire model is visible in the editor window.

It is also possible to pan over a model. Consider the window shown in Figure 2.28. Here, we have zoomed in so that icons are larger than the default. The **pan window** at the lower left shows the entire model, with a red box indicating the visible portion of the model. By clicking and dragging in the pan window, it is easy to navigate the entire model.

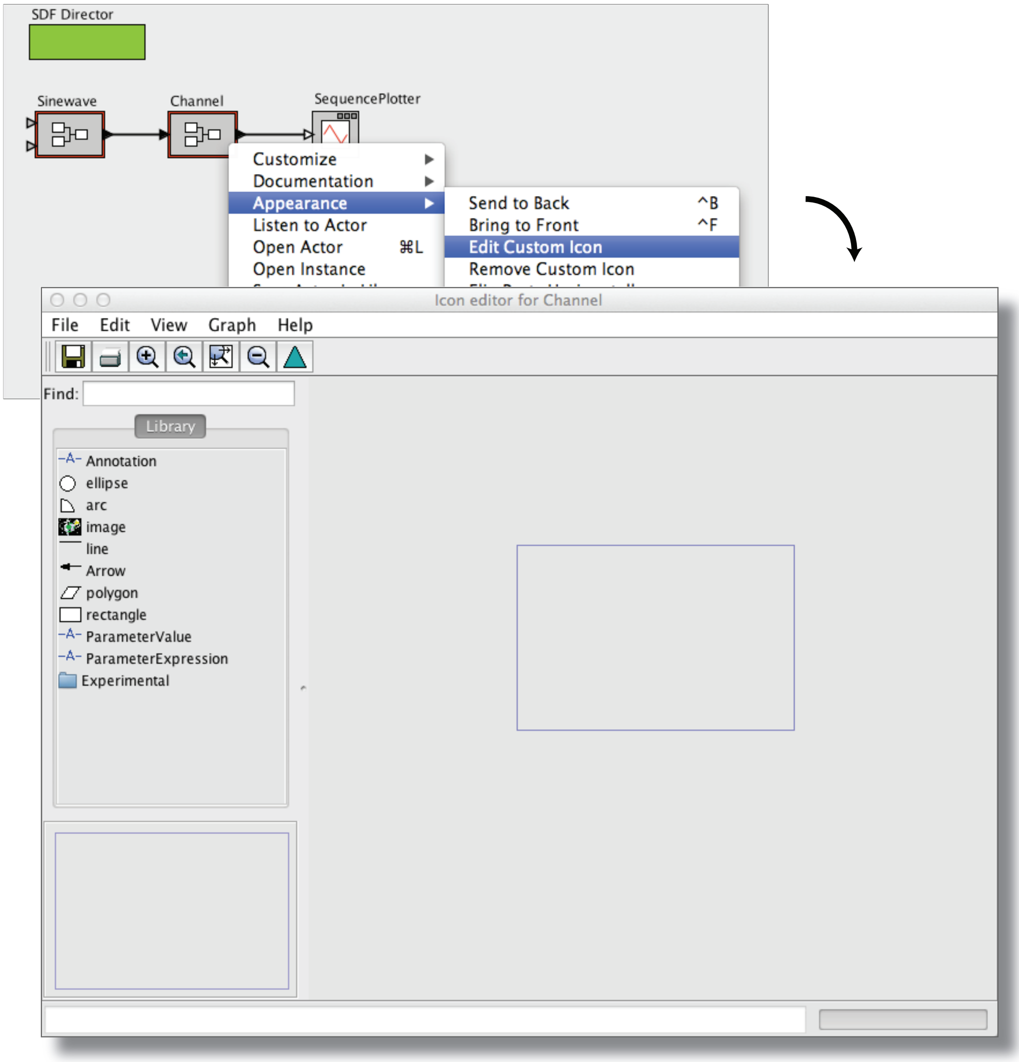


Figure 2.25: Custom icon editor for the Channel actor.

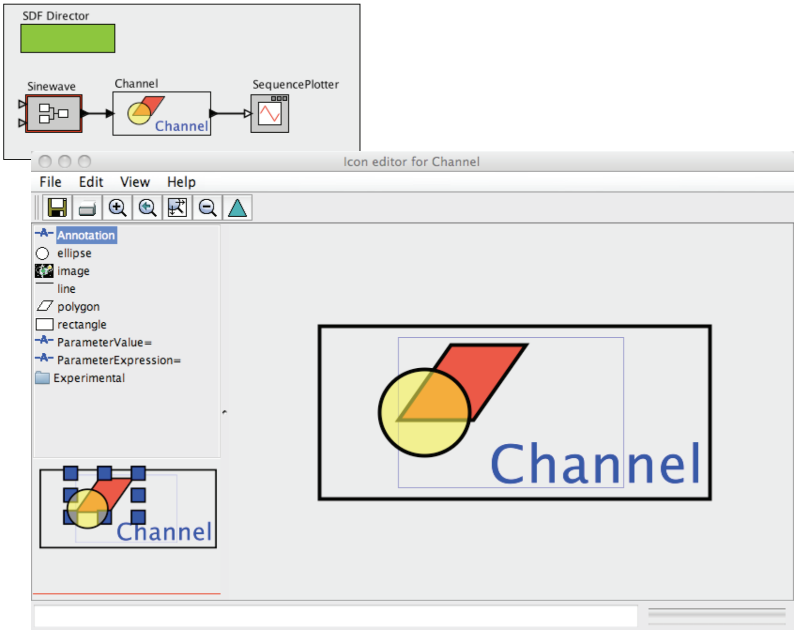


Figure 2.26: Custom icon for the Channel actor. [online]

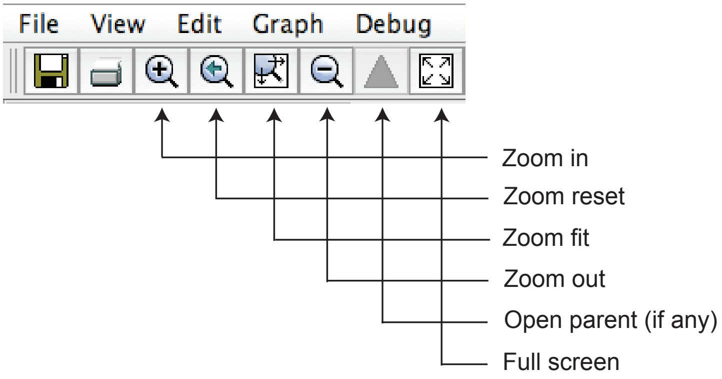
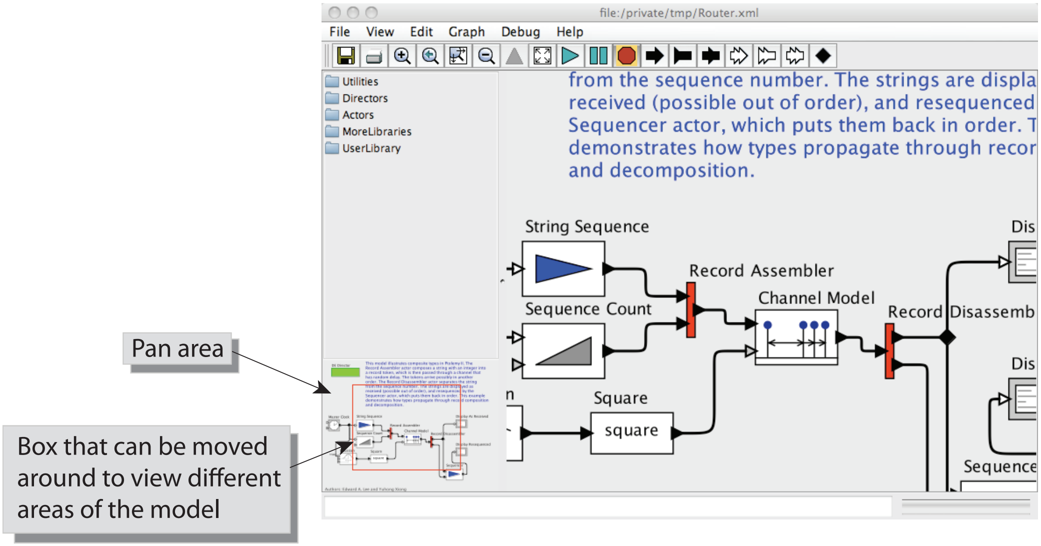


Figure 2.27: Summary of toolbar buttons for zooming and fitting.



from the sequence number. The strings are displa
received (possible out of order), and resequenced
Sequencer actor, which puts them back in order. T
demonstrates how types propagate through recor
and decomposition.

Figure 2.28: The pan window at the lower left has a red box representing the visible area of the model in the main editor window. This red box can be moved around to view different parts of the model.

2.6 Classes and Inheritance

Ptolemy II includes the ability to define **actor-oriented classes** (Lee et al., 2009a). These classes can then be used to create **instances** and **subclasses**, both of which use the concept of class **inheritance**. A class provides a general definition (or template) for an actor. An instance is a single, specific occurrence of that class, and a subclass is derived from the class — it includes the same structure, but may have some modifications. This approach improves design modularity, as illustrated in the example below.

Example 2.1: Consider the model that was developed in Section 2.3, shown for reference in Figure 2.29. Suppose that we wish to create multiple instances of the channel, as shown in Figure 2.30. In that figure, the sinewave signal passes through five distinct channels (note the use of a black-diamond relation between the sine wave and the channels to broadcast the same signal to each of the five channels). The outputs of the channels are added together and plotted. The result is a significantly cleaner sine wave than the one that results from one channel alone. (In communication systems, this technique is known as a diversity system, where multiple copies of a channel, each with independent noise, are used to achieve more reliable communications.)

Although it is functional, this is a poor design, for two reasons. First, the number of channels is hardwired into the diagram. (We will address that problem in the next section.) Second, each of the channels is a *copy* of the composite actor in Figure 2.29. Therefore, if the channel design needs to be changed, all five copies must be changed. This approach results in models that are difficult to maintain or scale.

A more subtle advantage to using classes and instances is that the XML file representation of the model will be smaller, since the design of the class is given only once rather than multiple times.

A better solution would be to define a Channel class, and use instances of that class to implement the diversity system. To implement this change, begin with the design in Figure 2.29, and remove the connections to the channel, as shown in Figure 2.31. Then right click and select `Convert to Class`. (Note that if you fail to first remove the connections, you will get an error message when you try to

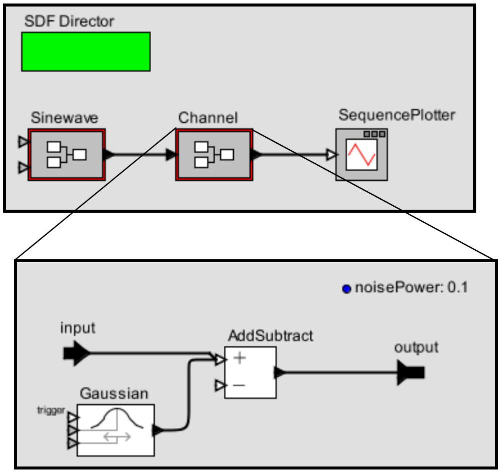


Figure 2.29: Hierarchical model that we will modify to use classes.

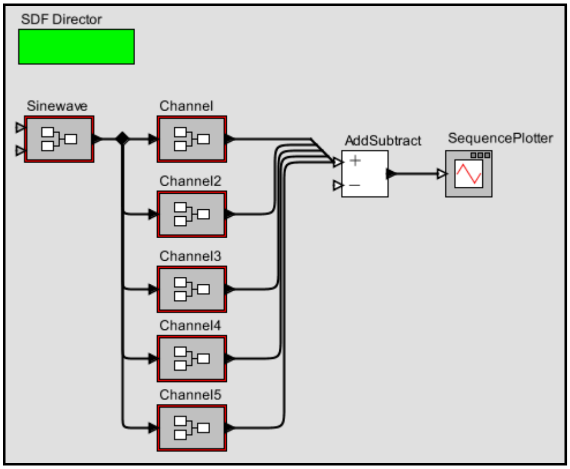


Figure 2.30: A poor design of a diversity communication system, which has multiple copies of the channel as defined in Figure 2.29. [online]

convert to class, as a class is not permitted to have connections.) The actor icon acquires a blue halo, which serves as a visual indication that it is a class rather than an ordinary actor (which is an instance).

Classes play no role in the execution of the model, and merely serve as definitions of components that must then be instantiated. By convention, we put classes at the top of the model, near the director, since they function as declarations.

Note also that instead of using `Convert to Class`, you can drag in an instance of **CompositeClassDefinition** from the `Utilities` library, and then select `Open Actor` and populate the class definition. You will want to give this class definition a more meaningful name, such as Channel.

Once you have defined a class, you can create an instance by right clicking and selecting `Create instance` or typing Control-N. Do this five times to create five instances of the class, as shown in Figure 2.31. Although this looks similar to the design in Figure 2.30, it is, in fact, a much better design, for the reasons described earlier. Try making a change to

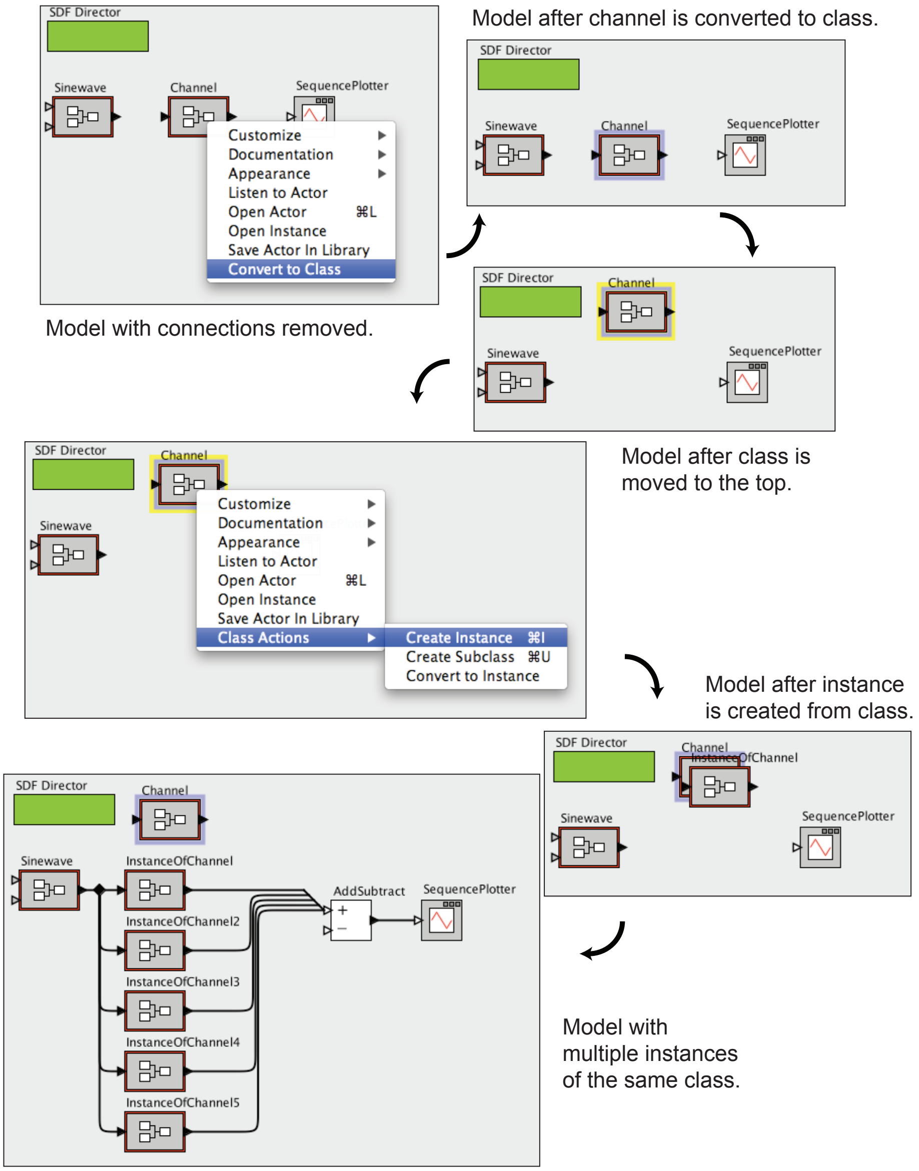


Figure 2.31: Creating and using a Channel class. [online]

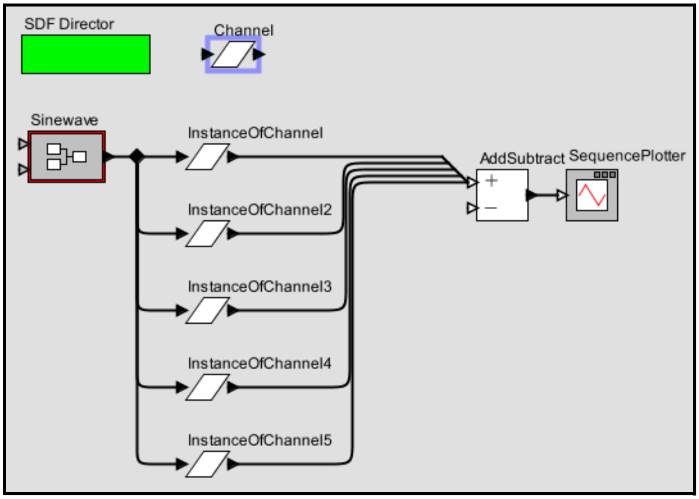


Figure 2.32: The model from Figure 2.31 with the icon changed for the class. Changes to the base class propagate to the instances. [online]

the class — for example, by creating a custom icon for it, as shown in Figure 2.32. Note that the changes propagate to each of the instances of the class.

If you invoke `Open Actor` on any of the instances (or the class) in Figure 2.32, you will see the same channel model. In fact, you will see the class definition. Any change you make inside this hierarchical model will be automatically propagated to all the instances. Try changing the value of the *noisePower* parameter, for example, and observe the result.

If you wish to view the instance rather than the class definition, you can select `Open Instance` on one of the instances. The window that opens shows only that instance. Each subcomponent that is inherited from the class definition is highlighted with a pink halo.

2.6.1 Overriding Parameter Values in Instances

By default, all instances of Channel in Figure 2.32 have the same icon and the same parameter values. However, each instance can be customized by overriding these values. In Figure 2.33, for example, we have modified the custom icons so that each has a different

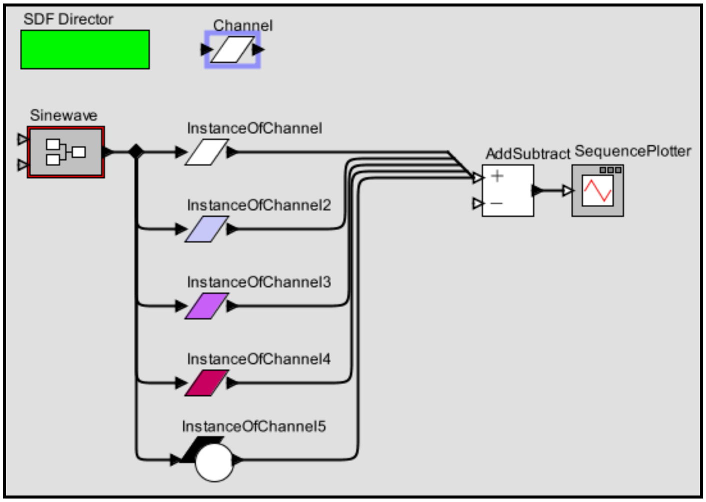


Figure 2.33: The model from Figure 2.32 with the icons of the instances changed to override parameter values in the class. [online]

color, and the fifth one has an extra graphical element. These changes are made by right clicking on the icon of the instance and selecting `Edit Custom Icon`. If you update class parameters that are overridden in an instance, then an update to the class will have no effect on the instance. It only has an effect on instances that have not been overridden.

2.6.2 Subclasses and Inheritance

Suppose now that we wish to modify some of the channels to add interference, in the form of another sine wave. A good way to do this is to create a subclass of the Channel class, as shown in Figure 2.34. A subclass is created by right clicking on the class icon and selecting `Create Subclass`. The resulting icon for the subclass appears on top of the icon for the class, and can be moved aside.

The subclass contains all of the elements of the class, but with the icons now surrounded by a pink halo. These elements are inherited and cannot be removed from the subclass (attempting to do so will generate an error message). You can, however, change their parameter values and add additional elements. Consider the design shown in Figure 2.35,

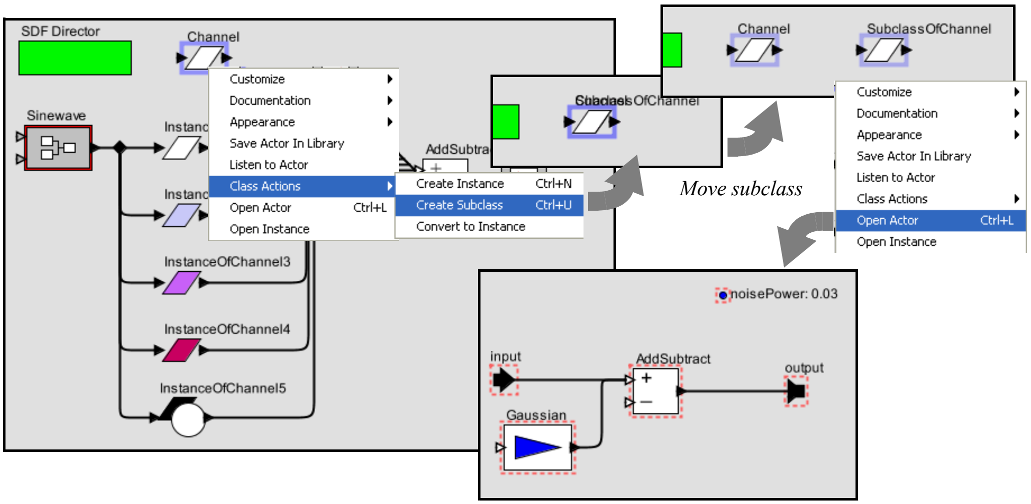


Figure 2.34: The model from figure 2.33 with a subclass of the Channel with no overrides (yet). [online]

which adds an additional pair of parameters named *interferenceAmplitude* and *interferenceFrequency* and an additional pair of actors implementing the interference.

A model that replaces the last channel with an instance of this subclass is shown in Figure 2.36, along with a plot showing the sinusoidal interference.

An instance of a class must be located in the same composite actor as the class itself, or in a composite actor that is itself contained in the model with the class itself (that is, in a submodel). To add an instance to a submodel, simply copy (or cut) an instance from the composite model that contains the class, and paste the instance into the submodel.

2.6.3 Sharing Classes Across Models

A class may be shared across multiple models by saving the class definition in its own file. We will illustrate this technique with the Channel class. First, right click and invoke `Open Actor` on the Channel class, and then select `Save As` from the `File` menu. The dialog that appears is shown in Figure 2.37. The check box labeled `Save submodel only` is

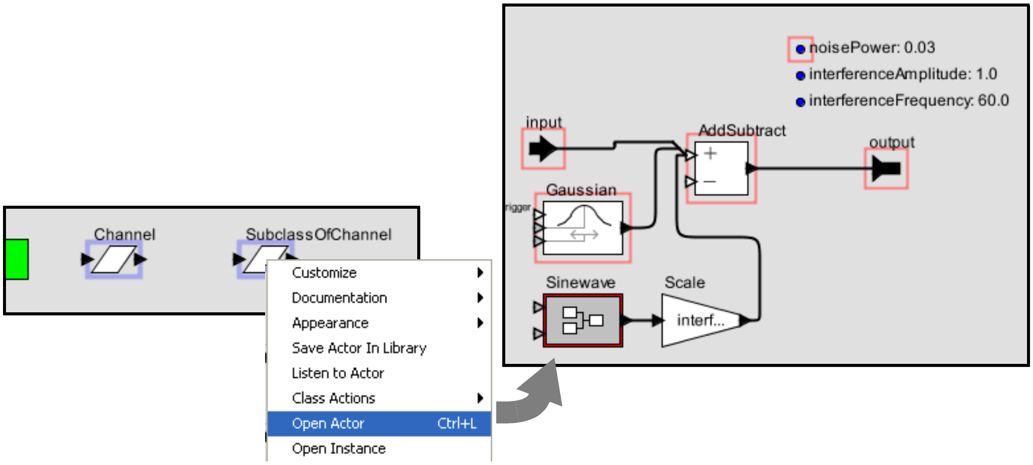


Figure 2.35: The subclass from Figure 2.34 with overrides that add sinusoidal interference. [online]

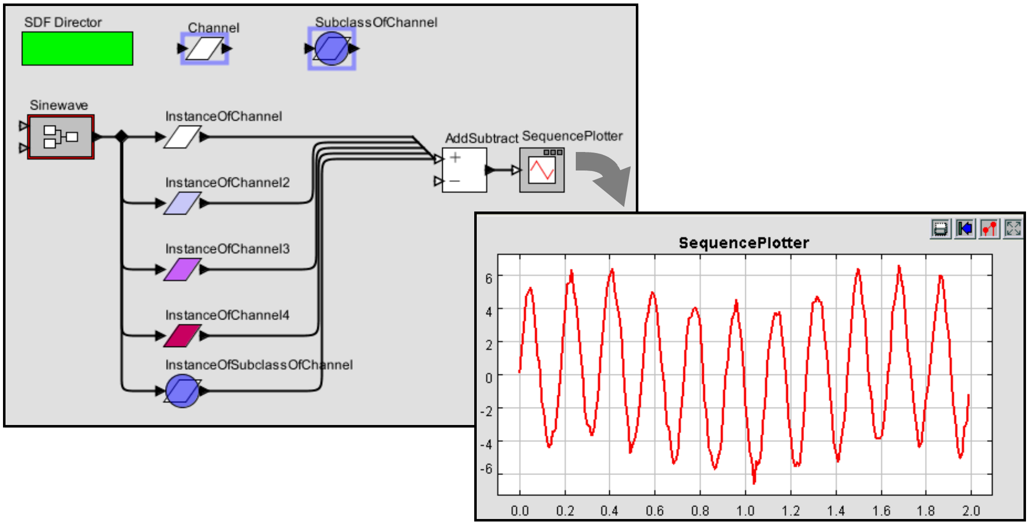


Figure 2.36: A model using the subclass from Figure 2.35 and a plot of an execution. [online]

by default unchecked. If left unchecked, the entire model will be saved.[6] In our case, we wish to save the Channel submodel only, so we must check the box.

It is important to save the class definition in a location that will be found by the model. In general, Ptolemy II searches for class definitions relative to the **classpath**, which is given by an environment variable called CLASSPATH. In principle, you can set this environment variable to include any directory that you would like to have searched. In practice, however, changing the CLASSPATH variable may cause problems with other programs, so we recommend that you store the file in a directory within the Ptolemy II installation directory or within a directory called ".ptolemyII" in your home directory.[7] In either case, Ptolemy will find class files stored in those directories.

Let's assume you save the Channel class to a file called `Channel.xml` in the directory `$PTII/myActors`, where `$PTII` is the location of the Ptolemy II installation. This class definition can now be used in any model, as follows. Open the model and select `Instantiate Entity` in the `Graph` menu, as shown in Figure 2.38. Enter the fully qualified class name relative to the `$PTII` entry in the classpath, which in this case is "`myActors.Channel`".

Once you have an instance of the Channel class that is defined in its own file, you can add it to the **UserLibrary** that appears in the library browser on the left side of Vergil windows, as shown in Figure 2.39. Right click on the instance and select `Save Actor in Library`. As shown in the figure, this causes another window to open and display the user library. The user library is itself a Ptolemy II model stored in an XML file. When you save the library model the class instance becomes available in the UserLibrary for any Vergil window (for the same user). Note that saving the class definition itself (vs. an instance of the class) in the user library does not have the same result. In that case, the user library would provide a new class definition rather than an instance of the class.

2.7 Higher-Order Components

Ptolemy II includes a number of **higher-order components**. These are actors that operate on the structure of the model rather than on input data. Consider the several examples

[6]On some platforms, a separate dialog will appear with the `Save submodel only` check box.

[7]If you don't know where Ptolemy II is installed on your system, you can invoke [`File`→`New`→`Expression Evaluator`] and type PTII followed by Enter. Or, in a Graph editor, select [`View`→`JVM Properties`] and look for the **ptolemy.ptII.dir** property.

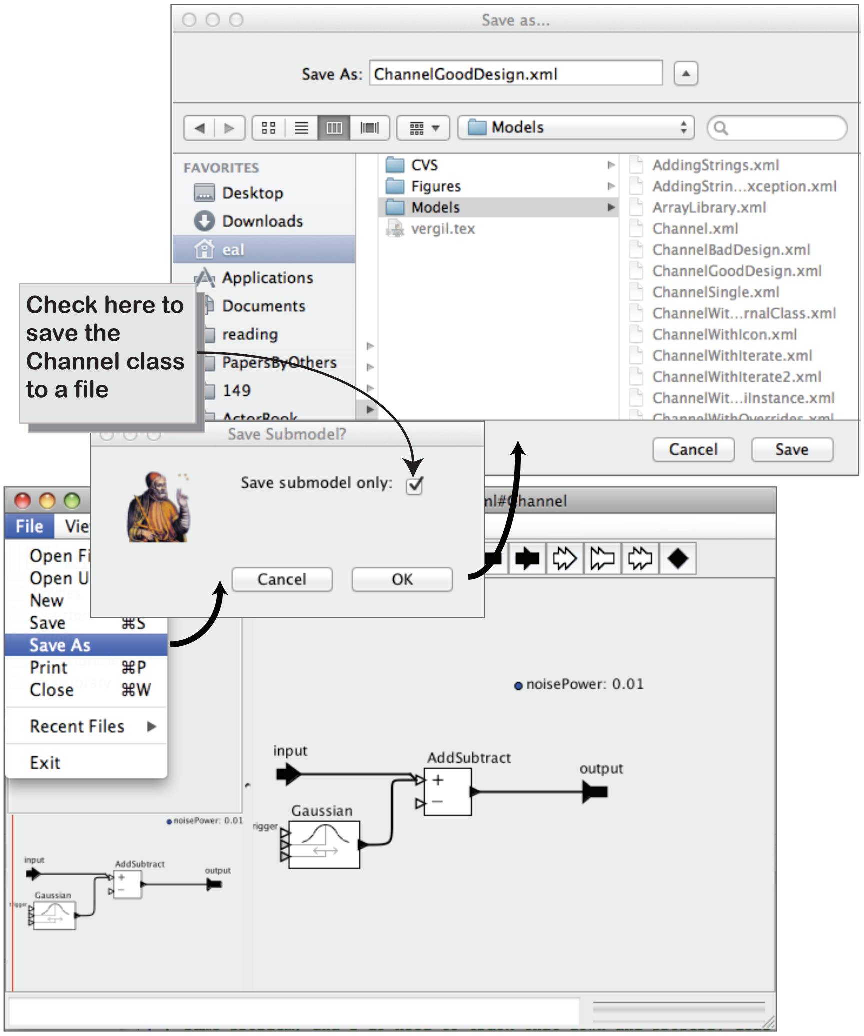


Figure 2.37: A class can be saved in a separate file to then be shared among multiple models. On some platforms, a separate dialog will appear with the `Save submodel only` check box, as shown. On others, the check box will be integrated into a single dialog. [online]

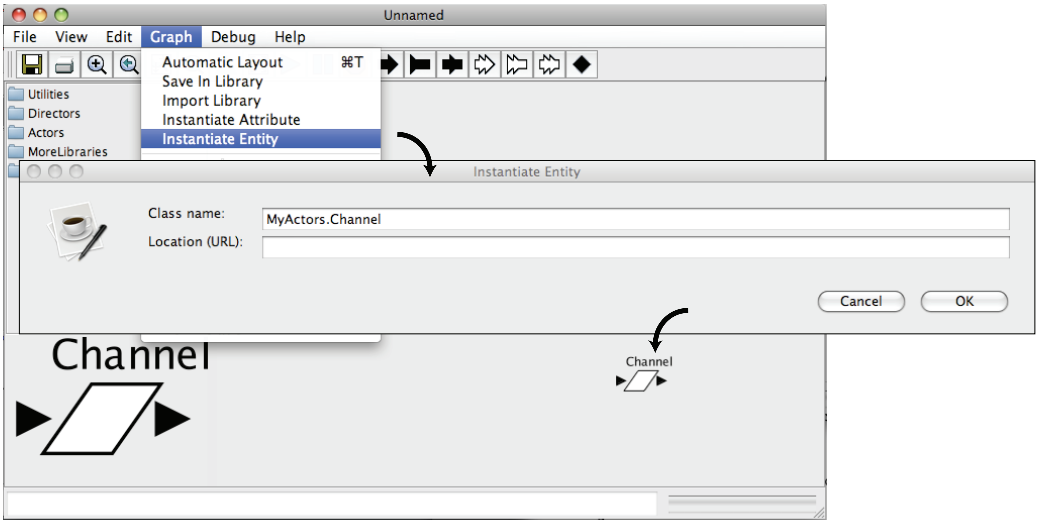


Figure 2.38: An instance of a class defined in a file can be created using `Instantiate Entity` in the `Graph` menu. [online]

above where five instances of the Channel actor are put into a model. Why five? Perhaps it would be better to have a single component that represents n instances of Channel, where n is a variable. This is exactly the sort of capability provided by higher-order actors. Higher-order actors, which were introduced by Lee and Parks (1995), make it easier to build large designs when the model structure does not depend on the scale of the problem. In this section, we describe a few of these actors, all of which are found in the `HigherOrderActors` library.

2.7.1 MultiInstance Composite

Consider the earlier model shown in Figure 2.32, which has five instances of the Channel class connected in parallel. The number of instances is hardwired into the diagram, and it is awkward to change this number, particularly if it needs to be increased. This problem can be solved by using the **MultiInstanceComposite** actor,[8] as shown in Figure 2.40. The MultiInstanceComposite is a composite actor into which we have inserted a single

[8]Contributed by Zoltan Kemenczy and Sean Simmons, of Research In Motion Limited.

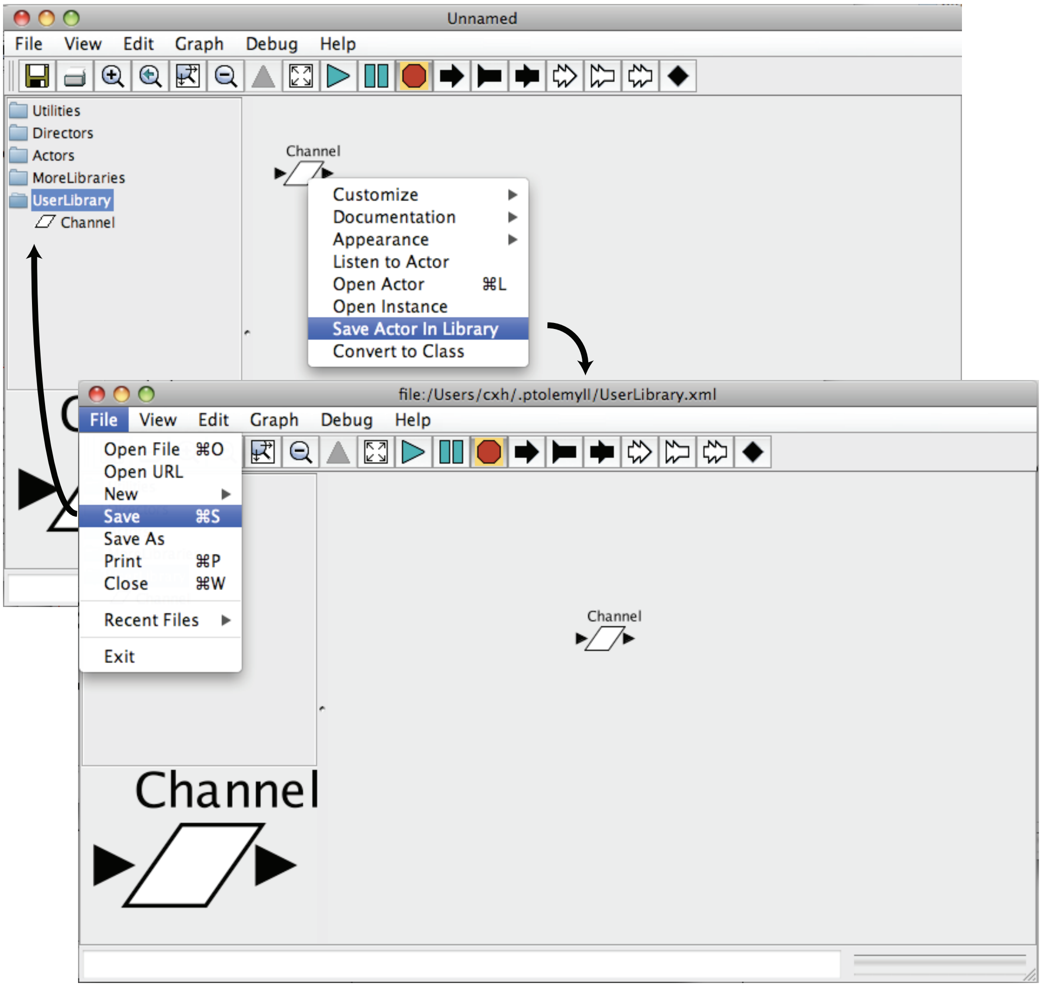


Figure 2.39: Instances of a class that are defined in their own files can be made available in the UserLibrary.

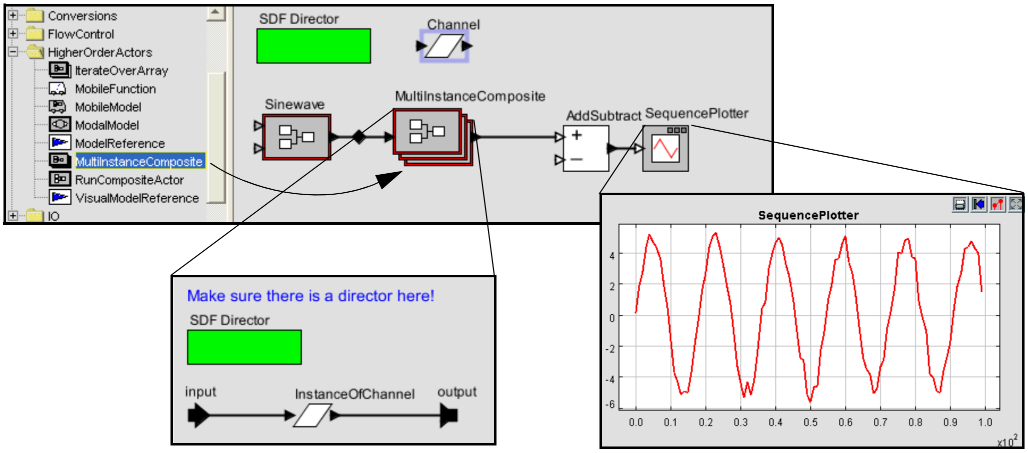
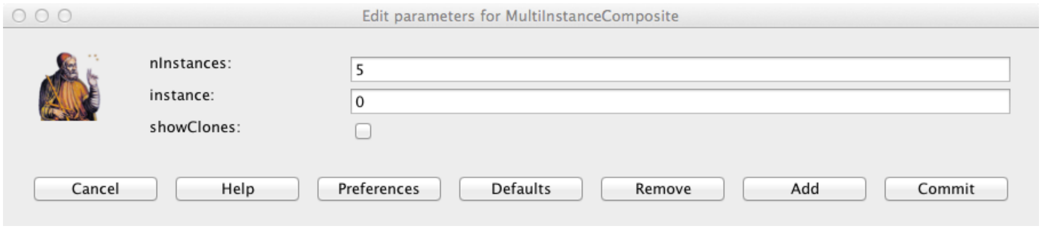


Figure 2.40: A model that is equivalent to that of Figure 2.32, but using a MultiInstanceComposite, which permits the number of instances of the channel to change by simply changing one parameter value. [online]

instance of the Channel (by creating an instance of the Channel, then copying and pasting it into the composite). MultiInstanceComposite is required to be opaque (meaning that it contains a director). It functions within the model as a single actor, but internally it is realized as a multiplicity of actors operating in parallel.

The MultiInstanceComposite actor has three parameters, *nInstances*, *instance*, and *showClones*, as shown in Figure 2.41. The first of these specifies the number of instances to create. At run time, this actor replicates itself this number of times, connecting the inputs and outputs to the same sources and destinations as the first (prototype) instance. In Figure 2.40, notice that the input of the MultiInstanceComposite is connected to a relation (the black diamond), and the output is connected directly to a multiport input of the AddSubtract actor. As a consequence, the multiple instances will be connected in a manner similar to Figure 2.32, where the same input value is broadcast to all instances, but distinct output values are supplied to the AddSubtract actor.

The model created using multi-instances is better than the original version because the number of instances can be changed with a single parameter. Each instance can be customized as needed by expressing its parameter values in terms of the *instance* parameter of the MultiInstanceComposite. Try, for example, making the *noisePower* parameter of



nInstances: 5
instance: 0
showClones: ☐

Figure 2.41: The first parameter of the MultiInstanceComposite specifies the number of instances. The second parameter is available to the model builder to identify individual instances. The third parameter controls whether the instances are rendered on the screen (when the model is run).

the InstanceOfChannel actor in Figure 2.40 depend on *instance*. For example, set it to `instance * 0.1` and then set *nInstances* to 1. You will see a clean sine wave when you run the model, because the one instance has number zero, so there will be no noise for that instance.

2.7.2 IterateOverArray

The implementation of the Channel class, which is shown in Figure 2.37, does not have any state, meaning that an invocation of the Channel model does not depend on data calculated in a previous invocation. As a consequence, it is not necessary to use *n* distinct instances of the Channel class to realize a diversity communication system; a single instance could be invoked n times on n copies of the data. This approach can be accomplished by using the **IterateOverArray** higher-order actor.

The IterateOverArray actor can be used in a manner similar to the MultiInstanceComposite in the previous section. That is, we can populate it with an instance of the Channel class, similar to Figure 2.40. The IterateOverArray actor also requires a director inside the model.

Example 2.2: Consider the example in Figure 2.42. In this case, the top-level model uses an array with multiple copies of the channel input rather than using a relation to broadcast the input to multiple instances of the channel.

This is accomplished using a combination of the Repeat actor (found in the `FlowControl→SequenceControl` sublibrary) and the SequenceToArray actor (see box on page 86). The Repeat actor has a parameter, *numberOfTimes*, which in Figure 2.42 is set equal to the *diversity* parameter. The SequenceToArray actor has a parameter *arrayLength* that has also been set to be equal to *diversity* (this parameter can also be set via the *arrayLength* port, which is filled in gray to indicate that it is both a parameter and a port). The output is sent to an ArrayAverage actor (see box on page 88).

The execution of the model in Figure 2.42 is similar to that of the earlier version, except that the scale of the output is different, reflecting the fact that the output is an average rather than a sum.

The IterateOverArray actor simply executes whatever actor it contains repeatedly on each element of an input array. The actor that it contains can be, as shown in Figure 2.42, an opaque composite actor. Interestingly, however, it can also be an atomic actor. To

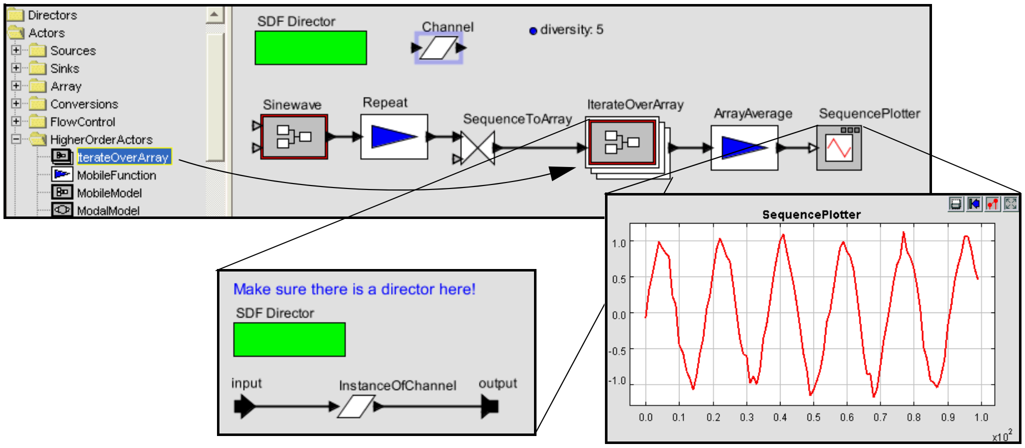


Figure 2.42: The IterateOverArray actor can be used to accomplish the same diversity channel model as in Figure 2.40, but without creating multiple instances of the channel model. This approach is possible because the channel model has no state. [online]

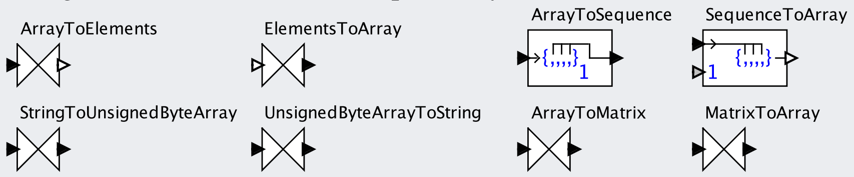
use an atomic actor with IterateOverArray, simply drag and drop the atomic actor onto an instance of IterateOverArray. It will then execute that atomic actor on each element of the input array, and produce as output the array of the results. This mechanism is illustrated in Figure 2.43. When an actor is dragged from the library and moved over the IterateOverArray actor, the icon acquires a white halo. The halo indicates that if the actor is dropped, it will be dropped into the actor under the cursor, rather than onto the model containing that actor. The actor you drop onto the IterateOverArray will become the actor that is executed for each element of the input array. In order for this to work with the Channel actor we defined above, however, we need to convert the Channel actor into an opaque actor by inserting a director, because IterateOverArray can only apply opaque actors to array elements.

2.7.3 Lifecycle Management Actors

A few actors in `HigherOrderActors` invoke the execution of a full Ptolemy II model. These actors generally associate ports (created by the user or actor) with parameters of the model. They can be used, for example, to create models that repeatedly run other mod-

Sidebar: Array Construction and Deconstruction Actors

The following actors construct and take apart arrays:



- **ArrayToElements** outputs the elements of an array on channels of the output port.
- **ElementsToArray** constructs an array from elements on channels of the input port.
- **ArrayToSequence** outputs the elements of an array sequentially on the output port.
- **SequenceToArray** constructs an array from a sequence of elements on the input port.
- **StringToUnsignedByteArray** constructs an array from a string.
- **UnsignedByteArrayToString** constructs a string from an array.
- **ArrayToMatrix** constructs a matrix from an array.
- **MatrixToArray** constructs an array from a matrix.

In addition, many polymorphic actors, such as AddSubtract, also operate on arrays.

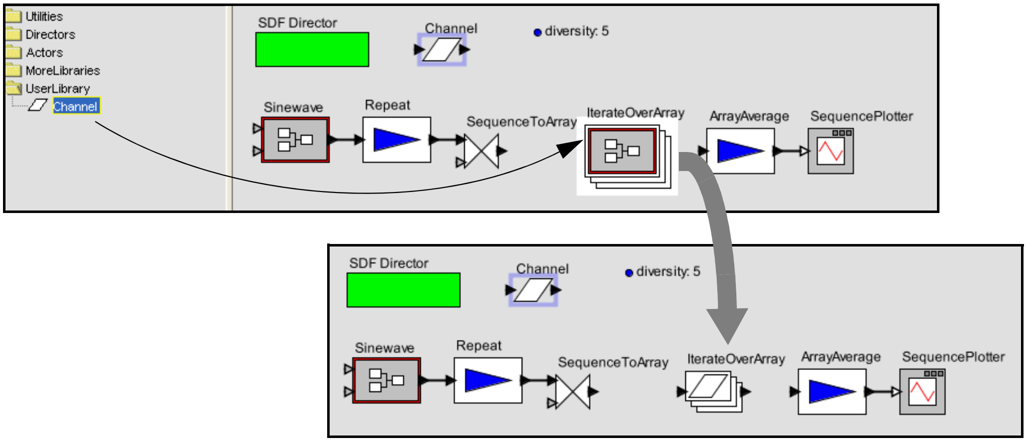


Figure 2.43: The IterateOverArray actor supports dropping an actor onto it. It transforms to mimic the icon of the actor you dropped onto it, as shown. Here we are using the Channel class, saved to the UserLibrary as shown in Figure 2.39.

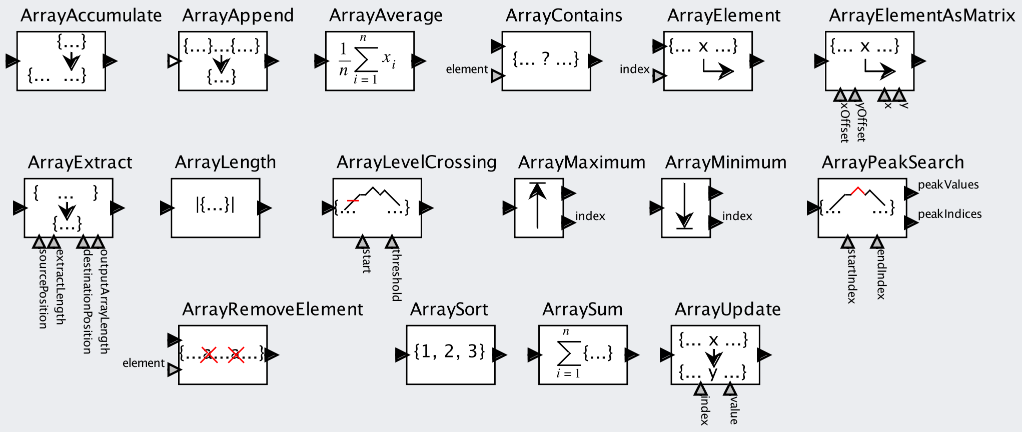
els with varying parameter values. These include **RunCompositeActor**, which executes the model that it contains. The **ModelReference** actor executes a model that is defined elsewhere in its own file or URL. The **VisualModelReference** actor opens a Vergil view of a model when it executes that model. Additional details can be found in the actor documentation and in the Vergil tour demonstrations.

2.8 Summary

This chapter has introduced Vergil, the visual interface to Ptolemy II, which supports graphical construction of models. Along the way, this chapter has introduced a number of capabilities of the underlying Ptolemy II system. Subsequent chapters will focus on properties of the various directors that are available. The appendices then focus on generic architecture and capabilities that span models of computation.

Sidebar: Array Manipulation Actors

The following actors operate on arrays:



- **ArrayAccumulate** appends input arrays, growing the output array.
- **ArrayAppend** appends input arrays provided on channels of a multiport.
- **ArrayAverage** averages the elements of an array.
- **ArrayContains** determines whether an array contains a specified element.
- **ArrayElement** extracts an element from an array.
- **ArrayElementAsMatrix** extracts an element using matrix-like indexing.
- **ArrayExtract** extracts a subarray.
- **ArrayLength** outputs the length of the input array.
- **ArrayLevelCrossing** finds an element that crosses a threshold.
- **ArrayMaximum** finds the largest element of an array.
- **ArrayMinimum** finds the smallest element of an array.
- **ArrayPeakSearch** finds peak values in an array.
- **ArrayRemoveElement** removes instances of a specified element.
- **ArraySort** sorts an array.
- **ArraySum** sums the elements of an array.
- **ArrayUpdate** outputs a new array like the input array, but with an element replaced.

In addition, many polymorphic actors, such as AddSubtract, also operate on arrays.

Sidebar: Mobile Code

A pair of actors in Ptolemy II support mobile models, where the data sent from one actor to another is a model to be executed rather than data on which a model operates. The **ApplyFunction** actor accepts a function in the expression language (see Chapter 13) at one input port and applies that function to data that arrives at other input ports (which you must create). The **MobileModel** actor accepts a MoML description of a Ptolemy II model at an input port and then executes that model, streaming data from the other input port through it.

A use of the ApplyFunction actor is shown in Figure 2.44. In that model, two functions are provided to the ApplyFunction in an alternating fashion, one that computes x^2 and the other that computes 2^x. These two functions are provided by two instances of the Const actor, found in the `Sources→GenericSources` sublibrary. The functions are interleaved by the Commutator actor, from the `FlowControl→Aggregators` sublibrary.

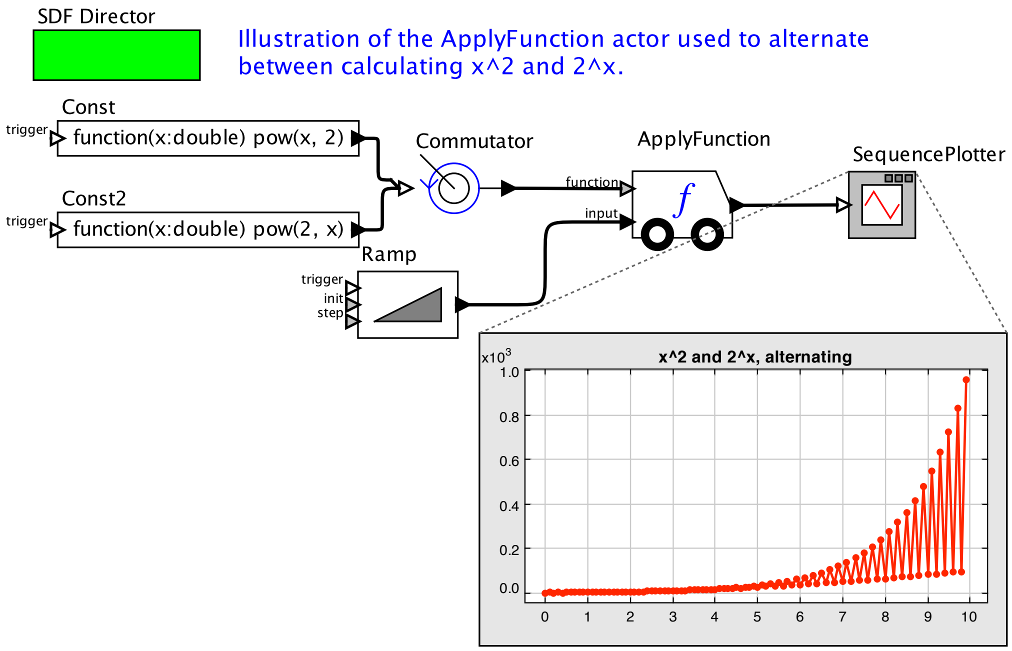


Figure 2.44: The ApplyFunction actor accepts a function definition at one port and applies it to data that arrives at the other port. [online]

Part II

Models of Computation

This part of this book introduces a few of the models of computation used in system design, modeling, and simulation. This is not a comprehensive set, but rather a representative set. The particular MoCs described here are relatively mature, well understood, and fully implemented. Most of these are available in other tools besides Ptolemy II, though no other tool provides all of them.