The Simulation of Synchronous Reactive Systems In Ptolemy II

by Paul Whitaker

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

UCB/ERL M01/20
May 2001

Abstract

The Synchronous Reactive (SR) domain in Ptolemy II models systems as components with both infinite processing resources and infinite communication bandwidth. Computation within components and data transfer between components are considered to happen instantaneously. Therefore, a given system is simulated as if all of its components execute and produce outputs at the same time. This raises interesting issues, particularly in systems with feedback. To guarantee a deterministic model of computation, outputs are permitted to have undefined value, and certain constraints are imposed on component behavior. Specialized components, such as finite state machines, may execute even when some inputs are undefined, producing only those outputs that can be determined from the defined inputs. This report outlines the semantics of the SR model of computation and describes its implementation as a domain in Ptolemy II.

Acknowledgements

First, I would like to thank Professor Edward A. Lee for introducing me to the Synchronous Reactive model of computation. His continuous support and guidance made this research possible. I am extremely grateful for the valuable advice he offered me regarding this report, and also over the course of my involvement with the Ptolemy project.

I would also like to thank Professor Thomas A. Henzinger for graciously consenting to act as a second reader of this report.

I am greatly indebted to Stephen Edwards for his work on Synchronous Reactive systems in Ptolemy Classic. I also appreciated his willingness to discuss his component execution scheduling algorithms. This paper stands on his shoulders.

I would like to thank all of my colleagues in the Ptolemy group for providing an excellent research environment and for helping me understand Ptolemy II. In particular, Jie Liu explained some of the implementation issues of fixed-point semantics. Xiaojun Liu taught me about finite state machines in Ptolemy II, allowing me to integrate them with my work. Stephen Neuendorffer clarified many fine points of the Ptolemy II kernel and sometimes helped me identify my mistakes. Yuhong Xiong answered all of my questions about the Ptolemy II type system. Christopher Hylands and Jörn Janneck reviewed a great deal of my work and provided invaluable programming advice.

Antonio Carrozza, Eric Schmidt, and Christy Wells read a draft of this report and offered suggestions.

This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of California MICRO program, and the following companies: Agilent Technologies, Cadence Design Systems, Hitachi, and Philips.

I would like to thank my family for their endless support and confidence in me. I owe them more than I could ever repay.

Throughout my work on this project, Anni Kolbaba was my inspiration. She gave me encouragement when I needed it most. This report is dedicated to her.

Table of Contents

1. Introduction
    - 1.1 Ptolemy II
    - 1.2 Models of Computation
    - 1.3 Hierarchical Heterogeneity
    - 1.4 Motivation for SR
2. Synchronous Reactive Semantics
    - 2.1 Overview
    - 2.2 Actor output monotonicity constraint
    - 2.3 Scheduling flexibility
    - 2.4 Actor prefire() monotonicity assumption
    - 2.5 Nonstrict actors
3. Synchronous Reactive Implementation
    - 3.1 Overview
    - 3.2 SR Receiver
    - 3.3 SR Randomized Scheduler
    - 3.4 SR Optimized Scheduler
    - 3.5 Handling of nonstrict actors
    - 3.6 SR Director
    - 3.7 Specialized actors
4. Synchronous Reactive Execution
    - 4.1 Simple Examples
    - 4.2 SR embedded in SR
    - 4.3 SR embedded in other domains
    - 4.4 Other domains embedded in SR
    - 4.5 FSM embedded in SR
    - 4.6 The Reflex Game
5. Conclusion
6. References

List of Figures

Figure 1. A screenshot from Vergil.
Figure 2. A co-dependent system.
Figure 3. An inconsistent system.
Figure 4. A system with multiple solutions.
Figure 5. A pointed complete partial order relating possible values in SR.
Figure 6. A scalar-valued CPO.
Figure 7. A vector-valued CPO.
Figure 8. A number of components can be grouped in a block.
Figure 9. A channel with two driving outputs.
Figure 10. A comparison of outputs to simulation convergence time.
Figure 11. A model to illustrate the prefire() monotonicity assumption.
Figure 12. Possible states and transitions of an SR Receiver.
Figure 13. A model with two valid and equally efficient possible schedules.
Figure 14. A system with feedback.
Figure 15. The dependency graph generated from Figure 14.
Figure 16. The strongly connected components of Figure 15.
Figure 17. The separation of a dependency graph into a head and a tail.
Figure 18. An equivalent representation of the graph in Figure 17.
Figure 19. A disconnected graph.
Figure 20. The execution results of Figure 19.
Figure 21. A model with a nonstrict three bit adder.
Figure 22. The execution results of Figure 21.
Figure 23. A second model with a nonstrict three bit adder.
Figure 24. The execution results of Figure 23.
Figure 25. A cyclic graph.
Figure 26. Another cyclic graph.
Figure 27. A model with instantaneous dialog.
Figure 28. The execution results of Figure 27.
Figure 29. A hierarchical SR model.
Figure 30. The same model, flattened.
Figure 31. A DT model containing an SR composite actor.
Figure 32. The contained SR model.
Figure 33. The execution results of Figure 31.
Figure 34. An FSM composite actor.
Figure 35. A model with two undefined inputs.
Figure 36. A model with one undefined input and one defined input.
Figure 37. A model with one defined input and one undefined input.
Figure 38. A model with two defined inputs.
Figure 39. The Reflex Game.
Figure 40. The RandomTime composite actor.
Figure 41. The SR Composite Actor.
Figure 42. The FSM Composite Actor.
Figure 43. The execution results of the Reflex Game.

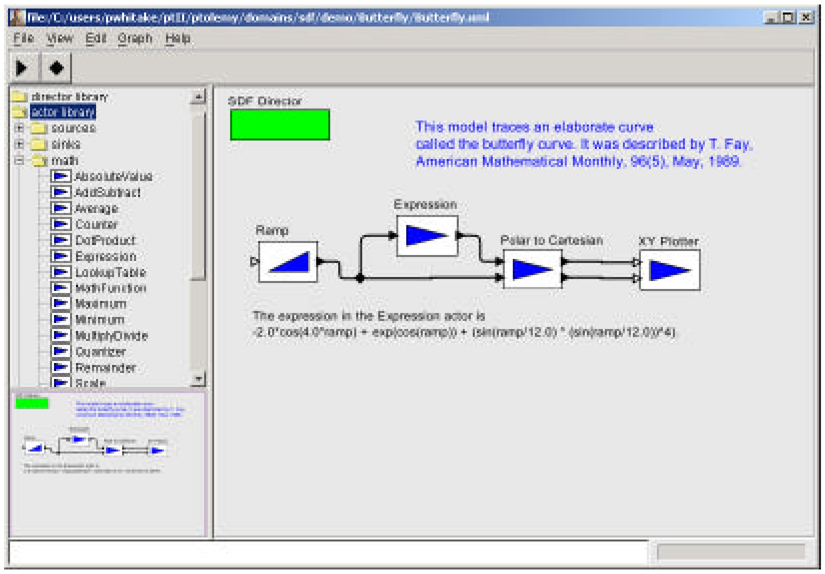
1. Introduction

1.1 Ptolemy II

The University of California at Berkeley Ptolemy project studies heterogeneous modeling, simulation, and design of concurrent systems. Ptolemy II [6], which is actively under development, is an environment for modeling heterogeneous, hierarchical systems. Computation in Ptolemy II is component-based, which makes it highly effective at simulating complex systems comprised of many different types of components. For example, in a single model, one component may perform signal processing, another may be responsible for feedback control, and another may produce three-dimensional graphics to represent the current state of the model.

Components called *actors* interact with one another by exchanging *tokens* of data via ports. During an execution of an actor (known as a firing), the actor may consume tokens from input ports, produce tokens on output ports, create visual output on the screen, access a file, or even produce audio output. Many actors have customizable *parameters*, making them quickly configurable for easy reusability.

Vergil is a front-end graphical user interface that allows users to create block diagrams that represent Ptolemy models. Users can drag and drop components, connect components, edit component parameters, and execute models. The visual syntax of Vergil is very appealing because model creation feels more like drawing than programming. It also offers an easy-to-read representation of component connections.



This model traces an elaborate curve
called the butterfly curve. It was described by T. Fay,
American Mathematical Monthly, 96(5), May, 1989.

The expression in the Expression actor is
-2.0*cos(4.0*ramp) + exp(cos(ramp)) + (sin(ramp/12.0) * (sin(ramp/12.0))^4)

Figure 1. A screenshot from Vergil.

1.2 Models of Computation

A *model of computation* [13] is a formal set of semantics that specifies the behavior and method of communication of components. A model of computation must specify a scheme for calculating a *schedule* (ordering) of component firings. It must define the manner in which time progresses (if there is a notion of time). It also must indicate how component communication occurs. Edward A. Lee, the director of the Ptolemy project, gives some guidelines [11] for choosing an appropriate model of computation for a specific purpose.

In Ptolemy II, a model of computation is realized as a *domain*. Domains currently implemented or under development in Ptolemy II are:

CSP	Communicating Sequential Processes
CT	Continuous Time
DDE	Distributed Discrete Event
DE	Discrete Event
DT	Discrete Time
FSM	Finite State Machines
Giotto	Time Triggered
GR	Graphics
HDF	Heterochronous Dataflow
PBO	Port-Based Objects
PDF	Parameterized Dataflow
PN	Process Networks
RTOS	Real-Time Operating Systems
RTP	Real-Time Processes
SDF	Synchronous Dataflow
SR	Synchronous Reactive

The domain in which components are executed determines many aspects of their behavior and interaction. This is the key that allows Ptolemy II to separate an abstract syntax from the semantics of a particular domain. Actors call methods of ports to send and receive data, but the domain specifies the behavior of these methods. Thus it is possible to write actors that will work in multiple domains, since an actor need not be aware of the mechanism for data transfer.

Some actors in Ptolemy II are *domain specific*. That is, upon firing, the actor makes certain assumptions about the domain, making the actor invalid in any domain that does not satisfy the assumptions. However, a careful effort is made to write *domain polymorphic* actors that will work in as many domains as possible, promoting reusability.

Likewise, some actors in Ptolemy II are *data specific*. They operate only on certain types of tokens. Other actors are *data polymorphic*, and can operate on any of a number of types of tokens. These actors may behave differently depending on the type of input data received. For example, an actor may add inputs that are numbers and concatenate inputs that are strings.

1.3 Hierarchical Heterogeneity

Often, the best way to describe a real-world system is by combining several different models of computation. A *composite actor* is an actor that can contain other actors. Composite actors are domain polymorphic since they can execute under many top-level domains. Since the actors in a composite actor can execute under a different set of semantics, different parts of a model can operate under different models of computation. Composite actors can also be arbitrarily nested. A composite actor can conveniently be considered a black box, so a user can analyze its behavior without necessarily understanding the details of the model inside.

The ability of Ptolemy II to integrate different models of computation into one simulation makes it *heterogeneous*. The ability of a Ptolemy II component to contain a sub-model makes it *hierarchical*. This hierarchical heterogeneity requires determining how different models of computation can interact, and it is a key focus of Ptolemy project research.

1.4 Motivation for SR

Benveniste and Berry [2] define a *Synchronous Reactive* system as one in which both computation within a component and communication between components happen instantaneously. In models with cycles, this introduces interesting issues involving instantaneous feedback. This also presents the possibility of instantaneous dialog, in which a component could query another component, receive a response, perform some computation, and produce an output, all with zero time delay.

The Synchronous Reactive (SR) model of computation offers precise control over the timing of events. Due to tight synchronization, SR is good at simulating systems with concurrent control logic. This same synchronization makes modeling events such as global resets or clock ticks a matter of ease.

In SR, some specialized actors can be executed multiple times in one instant, each time moving closer to a global solution. This means that certain models that would not be valid in other domains can produce meaningful results under SR semantics, which suggests that SR is better than many other domains for modeling certain systems.

2. Synchronous Reactive Semantics

2.1 Overview

A number of languages are based on the Synchronous Reactive model of computation. These synchronous languages include Esterel [3], Signal [1], Lustre [10], and Argos [15]. Ptolemy Classic [5] included an SR domain by Stephen Edwards [7], on which this Ptolemy II implementation is based.

Berry [4], the architect of Esterel, offers a synchrony hypothesis: assume that every reaction within a system is instantaneous. In SR, both data transfer between components and computation within components take no time. These events therefore must be atomic, which eliminates many complications from the semantics. For example, one need not determine an ordering of event beginnings or endings, since all events start and finish in the same instant. Thus, all components are considered to be executing and communicating simultaneously, and SR is a *synchronous* model of computation. An action of a component is instantaneous with respect to the environment, and also with respect to other components.

SR systems are *reactive* because they respond to inputs from the environment. Since the components are reactive, and computation within components is infinitely fast, output events are synchronized with input events. This makes SR different from many other models of computation, where component execution causes a time delay between the appearance of inputs and the production of outputs.

Using actions that occur instantaneously is a natural way to model reactive systems. In the real world, we consider objects in motion to react to each other instantaneously. We also regard our personal perception of our surroundings to be instantaneous.

Since inputs trigger outputs with absolutely no time delay, the presence of a feedback loop incites interesting issues.

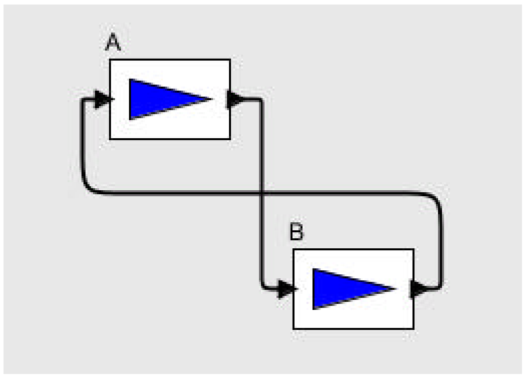


Figure 2. A co-dependent system.

In Figure 2, the output of component A is connected to the input of component B, and vice versa. The semantics of SR state that the two components should fire at the same time. Therefore, if actor A implements the function f_A and actor B implements the function f_B, then simultaneously, $f_A(x) = y$ and $f_B(y) = x$. It is not immediately clear how to execute such a system.

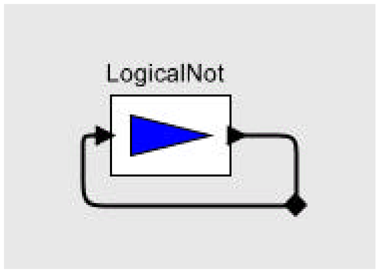


Figure 3. An inconsistent system.

The system in Figure 3 connects a negated value to the value itself. It is unclear how such a paradoxical system should behave.

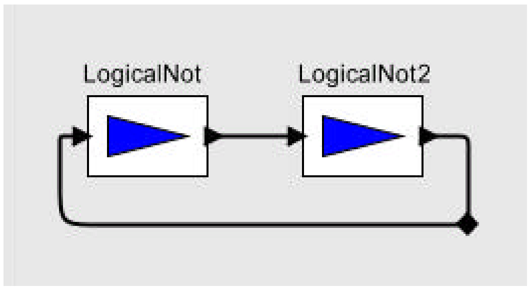


Figure 4. A system with multiple solutions.

In Figure 4, a value is negated twice and connected to the value itself. Assuming that the value is a boolean, two reasonable solutions for the value exist, and both are equally valid. It is impossible to determine the results of the execution of this system.

All of these matters will ultimately be resolved by permitting the value of a channel to be unknown. This solution, however, does not come without a price: for some models, certain values may never become known. Figures 3 and 4 are both such models.

2.2 Actor output monotonicity constraint

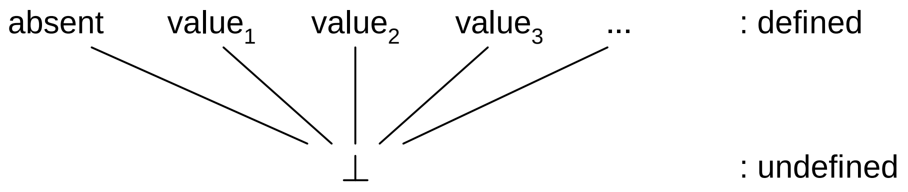


Figure 5. A pointed complete partial order relating possible values in SR.

Figure 5 shows the relationship between possible values in SR. The undefined element, which is represented by the symbol $\perp$ (pronounced "bottom"), is lower than every other element in the ordering. The relation, however, does not specify any means of comparing any other two elements in the set. Thus, any two defined elements in the partial order are incomparable.

The partial ordering is important because it can be used to define valid value transitions. To simulate an instant of an SR model, first the values on all channels must be set to the undefined value. Components can then execute and produce outputs, with the restriction that the new value must be higher than the old value in the partial order. One value is not permitted to change to another if the two are incomparable. Consequently, the only valid transition is for $\perp$, the undefined value, to become defined. This principle is the *actor output monotonicity constraint*, and all actors must obey it with each output they produce.

Each output can transition only once in an instant from unknown to some known value, so a model with *n* outputs can undergo at most *n* transitions in an instant. This seemingly insignificant observation leads to an important property of the model of computation: given a finite number of component outputs, an instant of the execution of the system can be simulated with a finite number of transitions. Therefore, any finite number of execution instants of the system can be modeled in a finite amount of time.

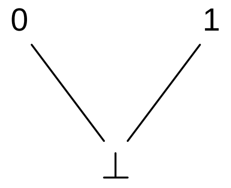


Figure 6. A scalar-valued CPO.

Figure 6 shows a complete partial ordering for a system with a single output. For simplicity, assume that the only valid values are 0 and 1. In each instant, this system will experience at most a single transition. The value will change from $\perp$ to 0 or from $\perp$ to

1. It is possible that no transition will occur in an instant, and the output will not change from ⊥. Thus, the execution of an instant could have three possible results.

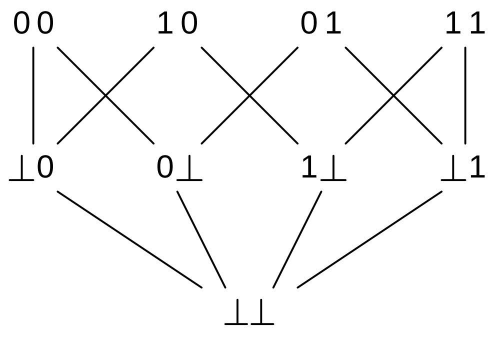


Figure 7. A vector-valued CPO.

The complete partial ordering in Figure 7 is for a system with two outputs. Once again, for the sake of simplicity, assume that the only valid values are 0 and 1. This system will undergo at most two transitions in a particular instant. Upward lines on the CPO represent the valid transitions. Note that certain results can be obtained in more than one manner; the order of a sequence of transitions does not affect the final state. If less than two transitions occur, either or both of the values could remain undefined.

It should be evident that the depth of the CPO for a system is simply the number of outputs. Since the CPO is finite, the maximum number of transitions also must be finite. Since transitions can only occur in the upward direction, component outputs implement *monotonic* functions. The step-size of each transition is fixed, so the system must eventually reach one node of the CPO and remain there. Therefore, the system is guaranteed to converge to a solution in finite time. The strategy for simulating an instant in SR is to execute components until the system reaches a stable state such that further execution of components would not result in further transitions. This stable state is known as a *fixed-point*. Executing one instant essentially amounts to finding a vector-valued solution to an equation.

The terms *known* and *defined* are used almost interchangeably, as are *unknown* and *undefined*. However, a subtle distinction exists. Typically, during the simulation of an instant, a value is referred to as known or unknown. It is always possible for a value that is unknown at a particular point in the simulation to become known by the end of the execution of the instant. If, after the simulation of the instant is complete, a value remains unknown, it is referred to as undefined.

Channels are not permitted to take multiple values within an instant, even if these values would approach a fixed-point. Such behavior would require metrics on data values to determine whether the values were converging. However, the single-value specification does not require that data values be measurable. This allows SR to be used without difficulty even when the best metrics on data are not clear.

2.3 Scheduling flexibility

In SR, all events happen simultaneously. Unfortunately, it is not possible to build a computer that could execute all components of a system simultaneously in a deterministic manner. Therefore, the problem of determining the order in which to fire components is "solved" by not specifying any scheme. In fact, with certain restrictions (to be described in a later section), components can be executed in any order and any number of times in a given instant. Thus for any particular model, many possible valid schedules exist. Schedulers for SR can handle disconnected graphs and cyclic graphs. Cyclic graphs make instantaneous feedback and instantaneous dialog possible.

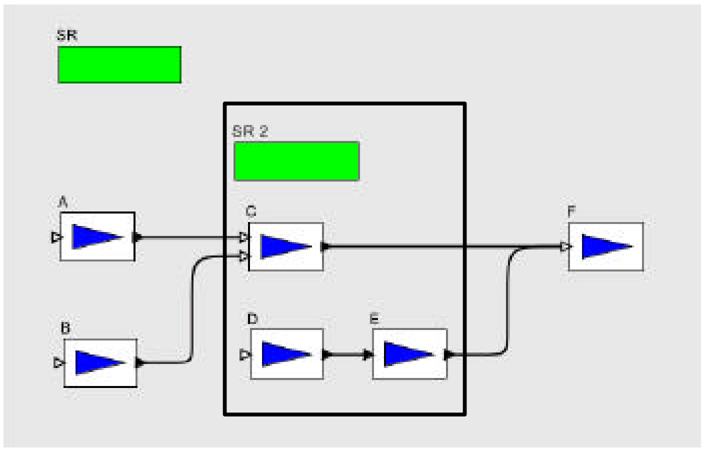


Figure 8. A number of components can be grouped in a block.

In addition to offering flexibility in scheduling, SR semantics also allow the application of certain graph transformations without modifying the behavior of the system. For example, SR is *compositional*. Any particular grouping of components can be hierarchically combined into an executable block without affecting the result of executing the entire system. SR is also *decompositional*. Blocks can be arbitrarily split and separately executed.

In most models of computation, components cannot be combined or separated without changing the semantics of the model. For example, such a transformation could create artificial deadlock in the Synchronous Dataflow (SDF) domain [12]. This is because the director for the SDF domain requires complete dataflow information to statically schedule models before execution. In SDF, determining a schedule ahead of time eliminates the processing overhead of dynamic scheduling. Edwards, in his thesis [7], presents an algorithm that exploits the decompositional properties of SR to achieve efficient predetermined scheduling of SR models.

Communication between components is unbuffered. In a particular instant, a given channel can have only one value, so there is no need for components to consume tokens of data. Thus any number of inputs can be connected to a particular channel without sacrificing determinacy. However, connecting multiple outputs to a single channel is not permitted, since it could cause unpredictable behavior.

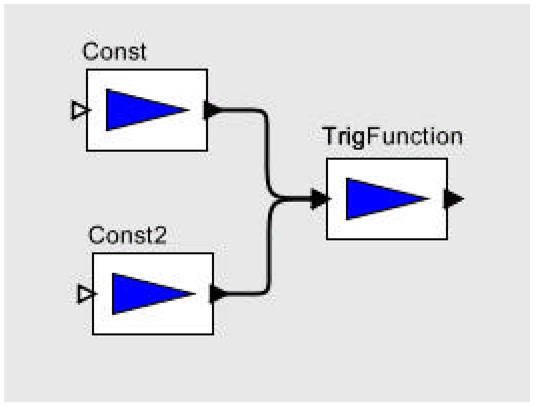


Figure 9. A channel with two driving outputs.

Only a single output is allowed to be connected to any particular channel. Otherwise, a race condition would occur, as shown in Figure 9. The value output to a particular channel would depend on the order of execution of components, and the model would not be deterministic.

Sequential component structure implies determinacy, but models with concurrently cooperating components can also be deterministic. For example, any number of components may send data to a component that displays inputs as a graph on the screen. Provided that the order the information is received does not affect the result, the model is deterministic.

The rate of convergence of an instant depends on the number of outputs in the model. In a best-case scenario, the components are executed in such a manner that at least one output becomes defined on each firing. Assume that the cost of firing each component is the same. If exactly one output becomes defined on each firing, the result of an instant will converge in a time linearly dependent upon the number of outputs.

In a worst-case scenario, one iterates through all of the components repeatedly, firing them each time. Each pass through the list of components, a random order is determined. In the very worst case, one fires all of the components that produce no output before, finally, the last component fired produces an output. If there are m components and n outputs, the n iterations through the list of m elements produce $n*m$ component firings. If we assume that every component has at least one output, then $n = m$, and there will be at most n^2 firings to simulate the instant.

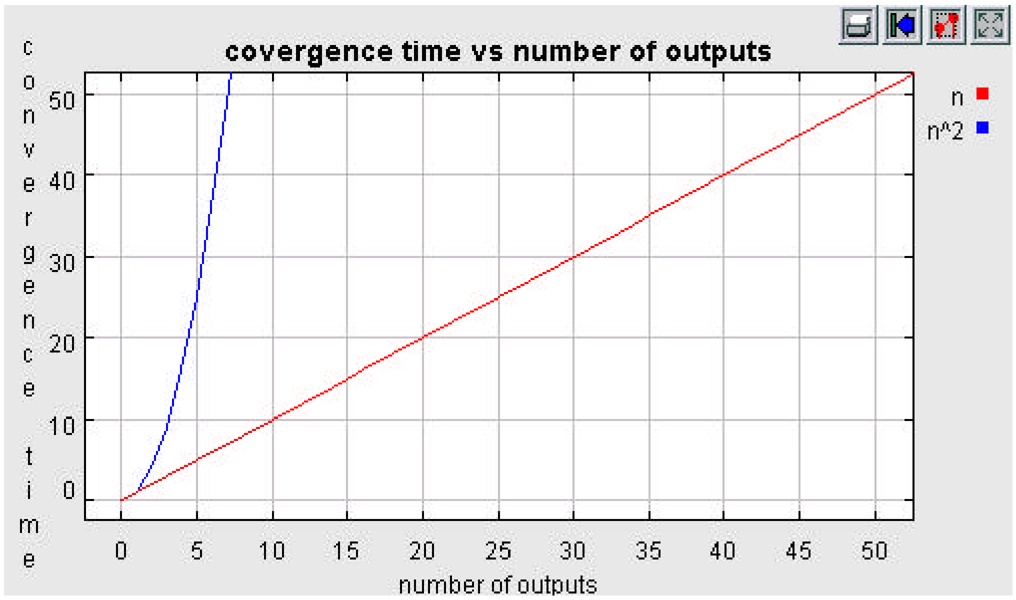


Figure 10. A comparison of outputs to simulation convergence time.

Thus, if there are *n* outputs in a model, the time of convergence is between *n* and n^2. Actual convergence rates can be expected to fall above the (lower) red line but below the (higher) blue line in Figure 10. This figure was actually created by an SR model in Ptolemy II.

It might be possible to generate extremely efficient schedules by using detailed information regarding the behavior of components. However, it is assumed that heterogeneity prevents detailed analysis of the internal functionality of components.

The scheduling flexibility of SR is quite convenient because the optimum scheduling strategy might depend on the application. For simulation, it is desirable to minimize the total computation time by minimizing the average time to execute an instant of the model. However, for implementation, it is required that the computation for any given instant completes in a certain period of time, and it is thus desirable to minimize the worst-case time to execute an instant. Any scheduling scheme that satisfies the semantics of SR is valid.

2.4 Actor prefire() monotonicity assumption

In Ptolemy II, an *iteration* of an actor consists of one call of the *prefire()* method, some number of calls of the *fire()* method, and one call of the *postfire()* method. The prefire() method returns true if the actor desires to be fired, and false otherwise. For example, an actor may return false in prefire() if there are insufficient inputs to act upon. The postfire() method returns true if the actor desires to be further iterated, and false otherwise. An actor may return false in postfire() if it wishes to become inactive after a certain number of firings.

For an SR model to be deterministic, all actors must satisfy the *prefire() monotonicity assumption*: if the prefire() method of an actor would return true if it were to be called at some point during the simulation of an instant, the method must return true if it were to be called at any later point during the simulation of an instant. In other words, the value that would be returned by an actor's prefire() method can only change from false to true during the course of the execution of an instant. As established in section 2.2, the quantity of known information in the model can only increase during the execution of an instant. Therefore, this constraint is roughly equivalent to stating that an actor should never decline to fire if it would consent to fire with fewer (or the same number of) known inputs.

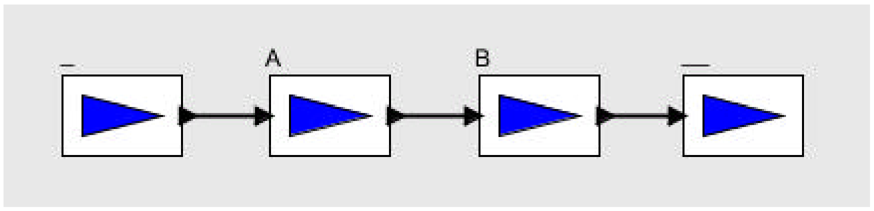


Figure 11. A model to illustrate the prefire() monotonicity assumption.

The following example describes a situation in which an actor would violate SR semantics by behaving contrary to the prefire() monotonicity assumption. In Figure 11, at the beginning of an instant, assume that the prefire() methods of both actors (if called) would return true, indicating willingness to fire. If actor A prefires and then fires, it may produce an input for actor B. If actor B violated the prefire() monotonicity assumption, it could now return false in prefire(), and, consequently, would not fire. However, if actor B prefires and fires first, then actor A will prefire and fire. In this case, unlike the first, both actors would execute. Therefore, if any actor violates the assumption, the behavior of the system would depend on the order of execution of components. Thus, to preserve scheduling flexibility and determinacy, we disallow actors that behave in such a manner. To explicitly verify that no actor violates this stipulation, it would be necessary to invoke and test the result of the prefire() method of all actors after any firing. Since this would consume considerable processor resources, the monotonicity of the prefire() method of each actor is assumed.

2.5 Nonstrict actors

Certain components may be able to produce one or more outputs even if some of their inputs are unknown. These components should be executed as *nonstrict actors*. Nonstrict actors may be fired any number of times in the simulation of an instant. Because values are not permitted to change once they are defined, a firing should not produce any outputs that cannot be definitively determined from the given known inputs. A component is not allowed to "change its mind" and output a different value later in the simulation of the instant. Therefore nonstrict actors must distinguish between inputs that are absent and inputs that are unknown.

Most non-SR components are considered *strict*. A strict component requires that all of its inputs are known before it is fired. Thus, a strict actor need not be aware that certain values might be unknown, because it will never be fired if any of its inputs are unknown, and therefore it will never perceive them. An SR model can contain any sub-model that operates according to a different model of computation as long as the sub-model is considered strict. This is convenient because it allows any model to be imported into SR as a strict block. Since a strict actor only executes when all of its inputs are known, there is no need to fire it multiple times in the same instant.

The handling of nonstrict components is a unique aspect of SR; most other models of computation do not distinguish between strict and nonstrict components. Consequently, using nonstrict actors in SR, it is possible to describe some systems that might be very difficult to specify using other models of computation, if not impossible.

Models of computation that can handle cyclic graphs require some means of breaking the dependency cycle. In other words, one of the components in the cycle must produce an output to start the computation of a loop. In SR, a nonstrict actor can achieve this goal, since a nonstrict actor may be able to produce some outputs even if some of its inputs are unknown.

3. Synchronous Reactive Implementation

3.1 Overview

In Ptolemy Classic, all components (called *stars*) were domain-specific. Creating a new domain required writing a set of stars to execute in the domain. The Ptolemy II SR domain was designed to be able to execute models containing existing domain-polymorphic actors. However, to take advantage of the domain's handling of nonstrict actors, these actors must be specially created.

In Ptolemy II, a domain defines a model of computation. Every domain has a *director* and a *receiver*. Each is implemented as a Java class. The director is responsible for regulating the execution of actors according to the semantics of the model of computation. The receiver is a receptacle for tokens of data. In a dataflow domain, for example, a receiver may implement a FIFO queue. In SR, however, the receiver is a mailbox that can contain only one token at a time.

Some domains, but not all, use a *scheduler*, which determines the order of actor firings. A scheduler is also implemented as a Java class. The Discrete-Event (DE) domain [16] is an example of a domain with no scheduler. Instead, actor firings are dynamically triggered by events, which are stored in a global queue. SR, however, has two schedulers. The SR Randomized Scheduler returns a random ordering of the actors that are controlled by the director. The director is expected to repeatedly fire the actors in this order until it determines that the values have reached a fixed-point. If the SR Optimized Scheduler is used, a single execution of the schedule guarantees that an instant converges to a fixed-point. Therefore, only one pass through the schedule is required, and the director need not explicitly check for convergence.

3.2 SR Receiver

The SR receiver is a mailbox with capacity one. A receiver can have known status, meaning it is either known to contain a token or known to not contain a token, or the status can be unknown. The isKnown() method of a receiver returns true if the receiver has known status. If a receiver has known status, the hasToken() method returns whether the receiver has a token. If a receiver has unknown status, calling its hasToken() method will result in an exception (specifically, an UnknownTokenException).

The get() method of a receiver returns the contained token if the receiver has known status and has a token. If the receiver has unknown status, an UnknownTokenException is thrown. If the receiver has known status but does not contain a token, a call to the get() method results in a ptolemy.actor.NoTokenException.

In the course of an iteration in SR, receivers can change from unknown status to known status, but never the other way around. The status of a receiver is automatically set to be known when the put() method is called to place a token in the receiver or the setAbsent() method is called to explicitly set the receiver to contain no token. Once a receiver becomes known, its value (or lack of a value if it is absent) cannot change until the next instant. The hasRoom() method of a receiver always returns true, indicating that it is possible to place a token in the receiver at all times. However, if a receiver has a value, and it receives a token that is not the same as the one it already contains, an exception of type IllegalOutputException will result. This exception will also be thrown if an actor attempts to place a token in a receiver that has absent status, or attempts to set a receiver to have absent status if it already contains a token.

Since the value of a receiver cannot change (once it is known) in the course of an iteration, tokens need not be consumed. A receiver retains its token until the director calls its reset() method at the beginning of the next iteration, which resets the receiver to have unknown status. Since the reset() method of a receiver is a protected method, there is no way for an actor to reset a receiver to have unknown status. The following table describes the informational methods of an SR Receiver. The left column indicates the current state of the receiver.

method / state	**isKnown()**	**hasToken()**	**get()**	**hasRoom()**
Unknown	False	UnknownTokenException	UnknownTokenException	True
Absent	True	False	NoRoomException	True
Value	True	True	Contained Value	True

Only the three action methods setAbsent(), put(), and reset() modify the state of an SR Receiver. The following table describes the action methods of an SR receiver:

method / state	**setAbsent()**	**put(Token)**	**reset()**
Unknown	Sets Absent	Puts Value	Sets Unknown
Absent	No Action	IllegalOutputException	Sets Unknown
Value	IllegalOutputException	*	Sets Unknown

(*) When a receiver contains a value and the put() method is called, no action results if the new value is the same as the old value. If the new value differs from the old value, an IllegalOutputException will result.

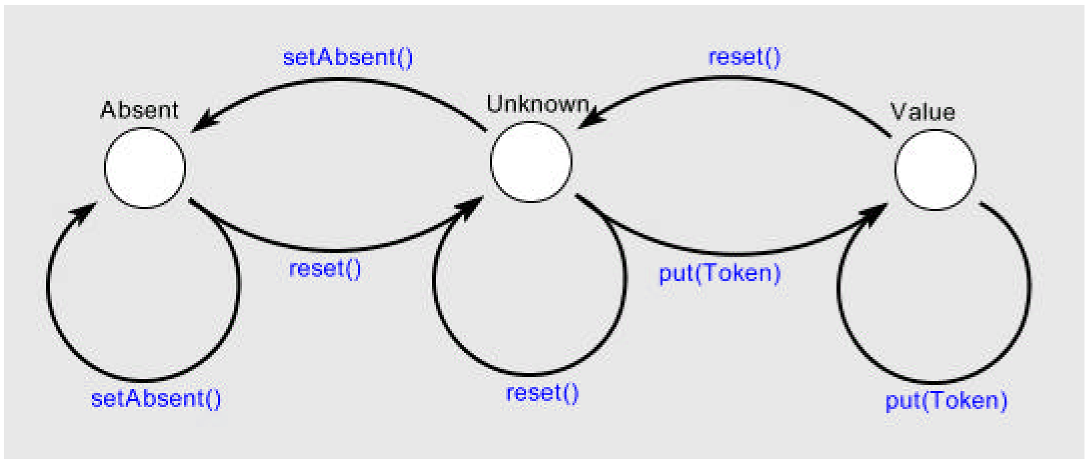


Figure 12. Possible states and transitions of an SR Receiver.

Figure 12 shows the states and state transitions of an SR Receiver. The receiver can enter the Absent state or the Value state from the Unknown state. The reset() method always returns the receiver to the Unknown state. Note that it is impossible for a receiver in the Absent state to directly transition to the Value state or vice versa.

3.3 SR Randomized Scheduler

The SR Randomized Scheduler produces a random ordering of the actors to be executed by the director. According to a strategy known as *chaotic iteration*, the director repeatedly executes the actors in this list, stopping when it determines that the system has reached a fixed-point. This scheduler is useful because it demonstrates that if all components meet the stated assumptions, SR semantics can be satisfied regardless of the execution order of individual components. It also seems to be a natural way to schedule firings, since there is no reason (other than efficiency of simulation) to choose any particular ordering.

3.4 SR Optimized Scheduler

Some domains, such as Synchronous Dataflow (SDF), are statically scheduled. At the cost of the preprocessing necessary to calculate a schedule, this eliminates the overhead of dynamic scheduling. For real-time systems, simulation speed can be very important, and static scheduling is highly desirable. A code generator sequences segments of machine code together to compute the results of component firings. If a schedule is available, these blocks of code can be appended in a predetermined order, reducing branch instructions and improving efficiency. This is particularly valuable for generating code for embedded systems. The SR Optimized Scheduler produces static schedules for the SR domain.

Acyclic models are straightforward to schedule, since components can be executed in topological order. An acyclic SR graph can be considered to be a dataflow model without computation or communication delay. An acyclic model can have multiple schedules that are all consistent with SR semantics. Figure 13 shows one such model.

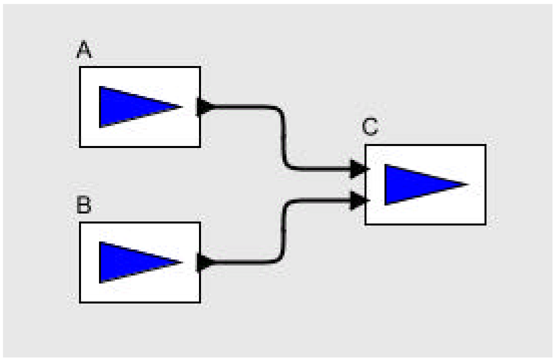


Figure 13. A model with two valid and equally efficient possible schedules.

In Figure 13, it is clear that an optimal schedule will execute both actors A and B before actor C. However, the ordering {A B C} and the ordering {B A C} are both valid and equally efficient.

The SR Optimized Scheduler implements an algorithm described by Stephen Edwards [7] for scheduling models that may have cycles. After the execution of schedule returned by this algorithm, further component firings would not result in more values becoming defined. Therefore, the schedule guarantees that the system will reach a fixed-point, so this approach will produce the same results as the chaotic iteration strategy.

The first step in the scheduling algorithm is to construct a directed *dependency graph* of the model. The nodes in the dependency graph are the output ports of all the actors to be executed. Thus, an actor with multiple outputs is represented by multiple nodes in the dependency graph. However, since we wish to execute actors that may have no outputs, we include a single node in the dependency graph to represent these actors. Edges are drawn from a first node to any second node that directly depends on the value of the first node. Figure 14 shows a model, and Figure 15 shows its dependency graph.

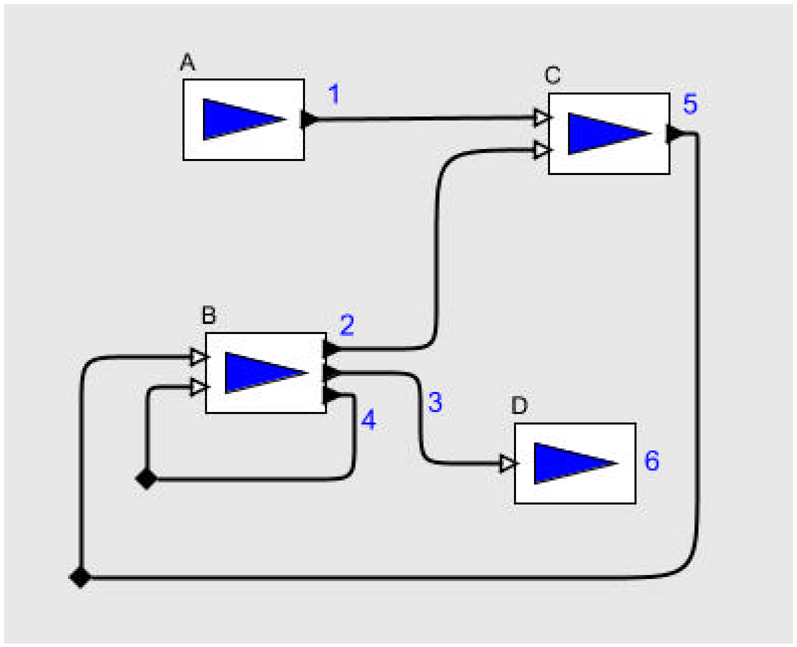


Figure 14. A system with feedback.

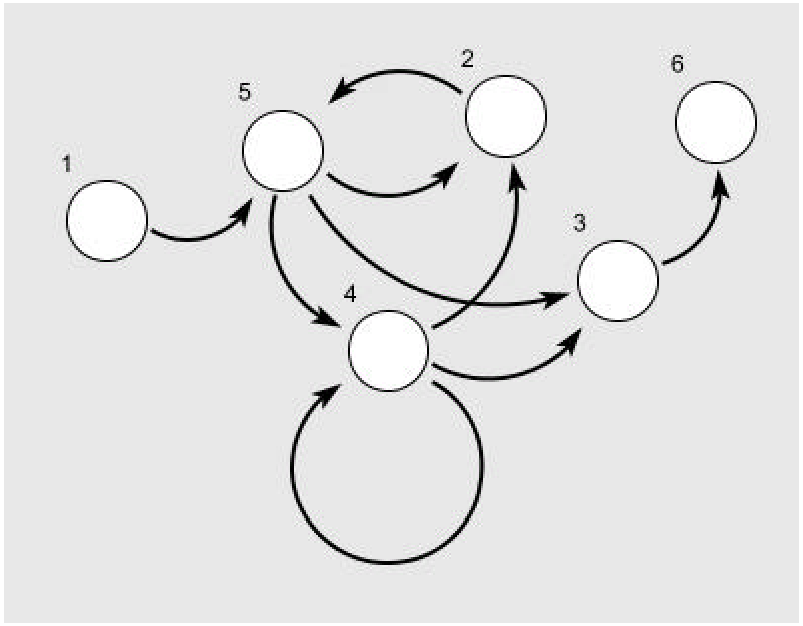


Figure 15. The dependency graph generated from Figure 14.

The *reachable set* of a node is the set of nodes that can be reached by starting at the node and following some path of edges. For example, in Figure 15, the reachable set of node 5 contains every node except node 1. A graph is *strongly connected* if every node in the graph is reachable from every other node in the graph. In other words, in a strongly connected graph, all nodes are mutually reachable. A *strongly connected component*, or SCC, is the largest subgraph of a graph such that the subgraph is strongly connected. The four strongly connected components from Figure 15 are shown in Figure 16. Three SCCs consist of a single node, and one SCC contains node 2, node 4, and node 5.

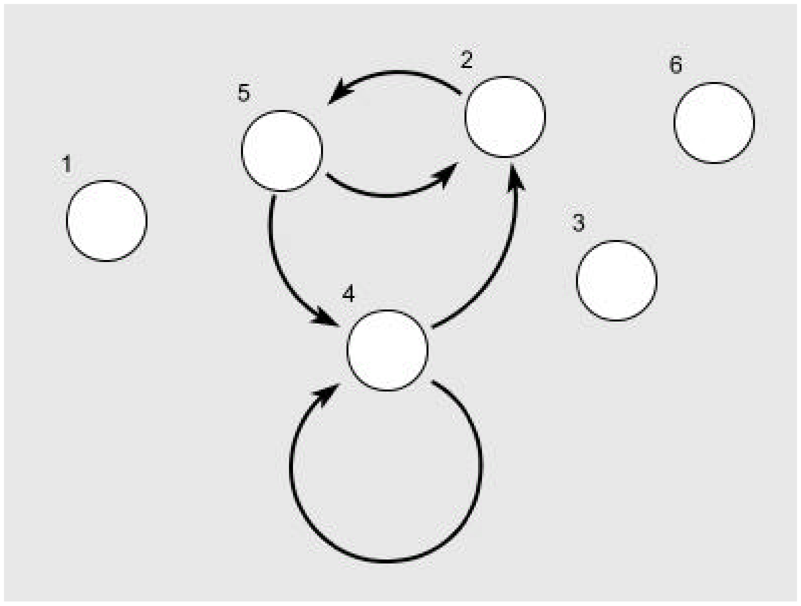


Figure 16. The strongly connected components of Figure 15.

In the original graph, the nodes in one SCC may be reachable from the nodes in another SCC. However, the converse must not be true, since that would make the two SCCs mutually reachable, and by definition, they would not be strongly connected components. Since the reachability relation is transitive and anti-symmetric, there must exist some ordering of SCCs such that any given SCC is not reachable from any other SCC that appears later in the ordering. In effect, this means that the strongly connected components of a graph can be sorted topologically.

In a dependency graph, we consider the nodes in the reachable set of a node to depend (either directly or indirectly) on the node. Since, in a topological ordering of SCCs, a given SCC does not depend on any subsequent SCC, schedules can be generated for each SCC and simply concatenated in this order.

If a subgraph consists of a single node, the schedule for the subgraph is simply one firing of the actor that corresponds to the node. However, the situation is significantly more complicated for SCCs that contain multiple nodes. Since all the nodes in an SCC are mutually reachable, all nodes in a given SCC are dependent on all other nodes in the SCC. Therefore, the SCC contains cycles, and its nodes cannot be topologically sorted.

To calculate the schedule of an SCC, we divide the nodes into a *head* and a *tail*. The dependency graph can now be represented in the form shown in Figure 17.

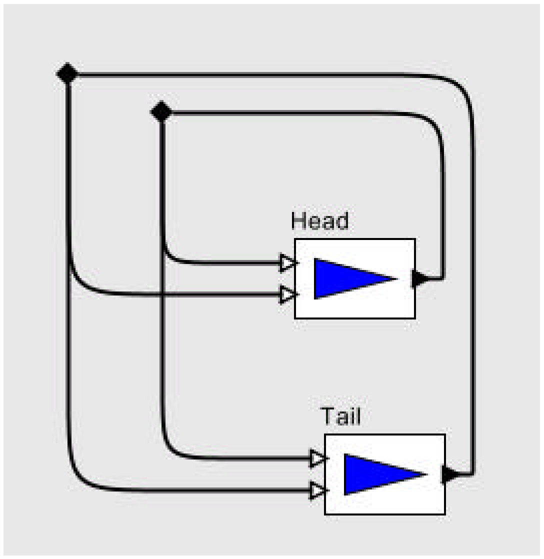


Figure 17. The separation of a dependency graph into a head and a tail.

This graph can also be represented in the slightly different, but topologically identical, form shown in Figure 18.

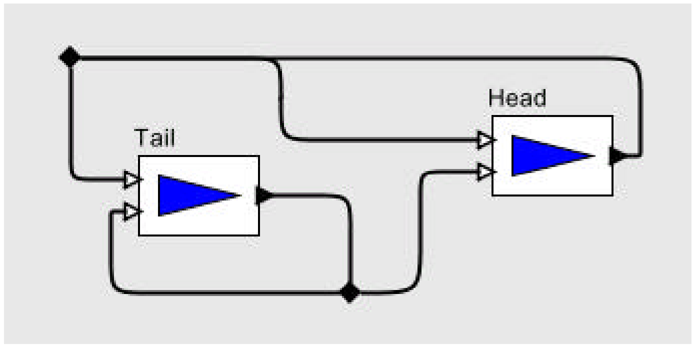


Figure 18. An equivalent representation of the graph in Figure 17.

Edwards [7] gives a formal proof that the head of the graph will converge to a fixed-point after *n* execution iterations of the tail and the head, where *n* is the number of nodes in the head. Once the fixed-point of the head is known, a final execution of the tail will bring the entire system to a fixed-point. Thus, the schedule returned is $(TH)^nT$, where *H* and *T* represent the schedules of the head and tail, respectively, and *n* signifies the number of repetitions of the parenthesized element. The schedules of the head and tail can be calculated by recursive application of this algorithm.

Some choices of head and tail separation produce better results than others. Since we have demonstrated that strongly connected graphs can be significantly more expensive to schedule, my implementation considers only separations that break strong connectivity in the tail. Given a strongly connected graph, strong connectivity can be broken by removing the predecessor set of any node. Since the node will have no predecessors, it will not be reachable from any other node. Therefore, the resulting graph will not be strongly connected. By choosing the head to be the predecessor set of a node, we achieve our goal of breaking strong connectivity in the resulting tail. My implementation also attempts to minimize the number of nodes in the head, since a large head can result in many more component firings. In Figure 16, the three single-node sub-graphs are trivial. In the other SCC, node 2 is the only predecessor of node 5. If we choose node 2 to be the head and remove it from the graph, the tail contains node 4 and node 5. The schedule of the head is *H* = {2}, and the schedule of the tail is *T* = {5 4}. Therefore, the schedule of the SCC is *THT* = {5 4 2 5 4}. To compute the schedule of the entire dependency graph, we concatenate the schedules of the SCCs in topological order, obtaining {1 5 4 2 5 4 3 6}. Finally, translating nodes in the dependency graph to corresponding actors, we obtain the schedule {A C B B C B B D}.

The SR Optimized Scheduler has several advantages over the SR Randomized Scheduler. It typically requires significantly fewer actor firings to arrive at a fixed-point. Instead of producing a schedule that may require repeated execution depending on dynamic data, the optimized scheduler generates a predetermined static schedule. It also eliminates the need for the director to explicitly determine whether the simulation has reached a fixed-point.

3.5 Handling of nonstrict actors

Unless an actor contains an attribute called "_nonStrictMarker", it is assumed to be a strict actor, meaning that it requires all of its inputs to be known before it is fired. This is very important since once an actor defines a particular output, it is not allowed to change that value in a subsequent firing in the course of the iteration. Thus, for a nonstrict actor to be valid in SR, a firing must produce the same outputs given the same inputs (in a given iteration). This is not an issue for strict actors, since they need only be executed once in the course of an instant.

In my implementation, actors need not have knowledge of whether their outputs have already been defined in the current instant. If an output channel has a value and an actor outputs a token with the same value, nothing happens. However, if an actor outputs a token of a different type or with a different value, the actor has attempted to perform an action not in accordance with SR semantics, and an exception is thrown.

3.6 SR Director

A director is responsible for determining an execution order for components, possibly obtaining this information from a scheduler. It then must fire the components, update time according to the semantics, and handle any idiosyncrasies of the particular domain. The director is typically the most complex element of a domain. The SR Director is no exception.

SR is an untimed domain, so it has no notion of the passage of time. Computation happens in a series of instants. An instant is one iteration of the director. If SR is embedded inside a timed domain, the SR Director will inherit the current time of the outside director.

Before each iteration, the director sets all receivers to the unknown state by calling the reset() method of each one. In my implementation of SR, receivers notify the director when they transition from the unknown state to a known state. The receivers inform the director of the change by calling the director's incrementKnownReceiverCount() method. Thus, the director need not query all of the receivers to determine the number with known state.

An actor is considered *ready to fire* if sufficient known inputs are available. Actor firings are essentially triggered by these known inputs since the director only fires an actor if it is ready to fire. An actor *has completed firing* if it has defined all of its outputs. It is assumed that a strict actor does not distinguish between absent and undefined values. Therefore, if a strict actor fires and leaves some of its outputs undefined, the director will assume that these values are absent, and explicitly make them so. There is no concern that the actor may attempt to change this value, since a strict actor may only be executed once in an instant.

An actor is considered *allowed to fire* if its prefire() method has returned true. Each time the director desires to execute an actor that is not allowed to fire, the prefire() method of the actor will be called to attempt to obtain permission. An actor is considered *allowed to iterate* if its postfire() method has not returned false. If the postfire() method of an actor does return false, the actor will not be fired in any subsequent iteration of the director.

The name of the scheduler to be used is specified by the *scheduler* parameter of the director. If the SR Randomized Scheduler is used, in the course of an iteration, the director repeatedly cycles through the schedule, firing those actors that are allowed to fire and ready to fire. An iteration *has converged* if both the total number of known outputs and the number of actors that are allowed to fire have converged. In other words, the director detects a fixed-point when all ready-to-fire actors are executed, and no new outputs are produced. Since further execution would not result in more defined outputs, the iteration ends. Note that all receivers having known state is not a sufficient condition for the convergence of an iteration. After the values are present, each actor must have the opportunity to fire before the end of the iteration.

If the SR Optimized Scheduler is used, the system arrives at a fixed-point after a single pass through the schedule, so the director need not check for convergence. The execution results are equivalent regardless of which scheduler is used.

The *iterations* parameter of the director determines how many instants of the system will be simulated. When the iteration count reaches this value, the postfire() method of the SR Director returns false to indicate that it does not wish to be fired again.

3.7 Specialized actors

I have created a number of nonstrict actors for use in the SR domain. Since directors for other domains would simply ignore the nonstrict marker attribute, these actors could be valid in other domains. However, their behavior is most interesting in SR. The following table describes these specialized actors:

Absent	Calls the sendAbsent() method of the output port to explicitly output an absent value
Undefined	Produces no token, so the output remains undefined
InstantaneousDialogGenerator	Generates an instantaneous dialog with another actor
NonStrictDelay	Delays tokens by one instant
NonStrictDisplay	Displays inputs if they exist, or otherwise indicates whether they are absent or unknown
NonStrictLogicFunction	Performs a logical function (*and*, *or*, *xor*, *nand*, *nor*, or *xnor*) on the inputs, producing a result if it can be determined from the known inputs
NonStrictSelect	Selects from the input channels based on a control input
NonStrictThreeBitAdder	Adds three input bits, producing a result if it can be determined from the known inputs

4. Synchronous Reactive Execution

4.1 Simple Examples

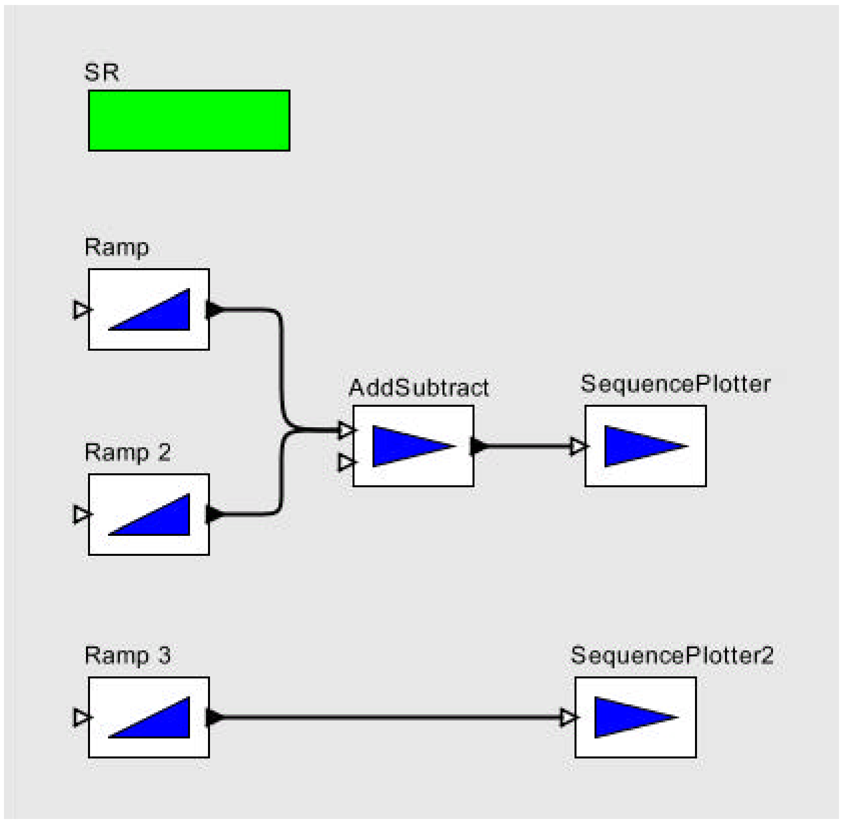


Figure 19. A disconnected graph.

As mentioned, SR is able to schedule disconnected graphs for execution. Figure 19 shows one such graph. Two ramps are connected to the plus port of AddSubtract, and the result is plotted. A third ramp is also plotted. Figure 20 shows the results of its execution in Ptolemy II.

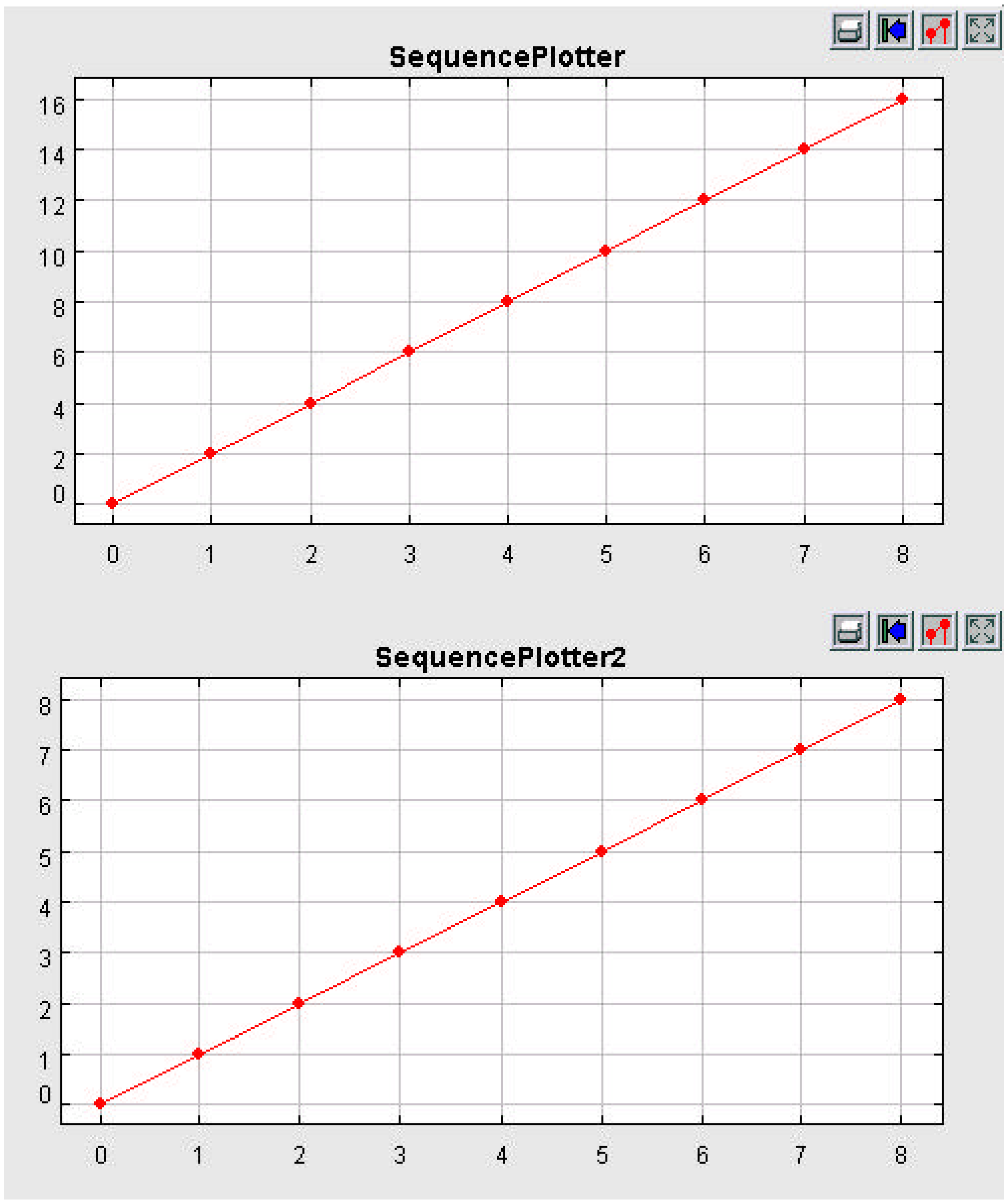


Figure 20. The execution results of Figure 19.

Figure 21 shows a model including a three bit adder which is nonstrict. The only valid inputs to the adder are 0 and 1. The actual inputs to the adder are a constant (1), another constant (0), and an undefined value.

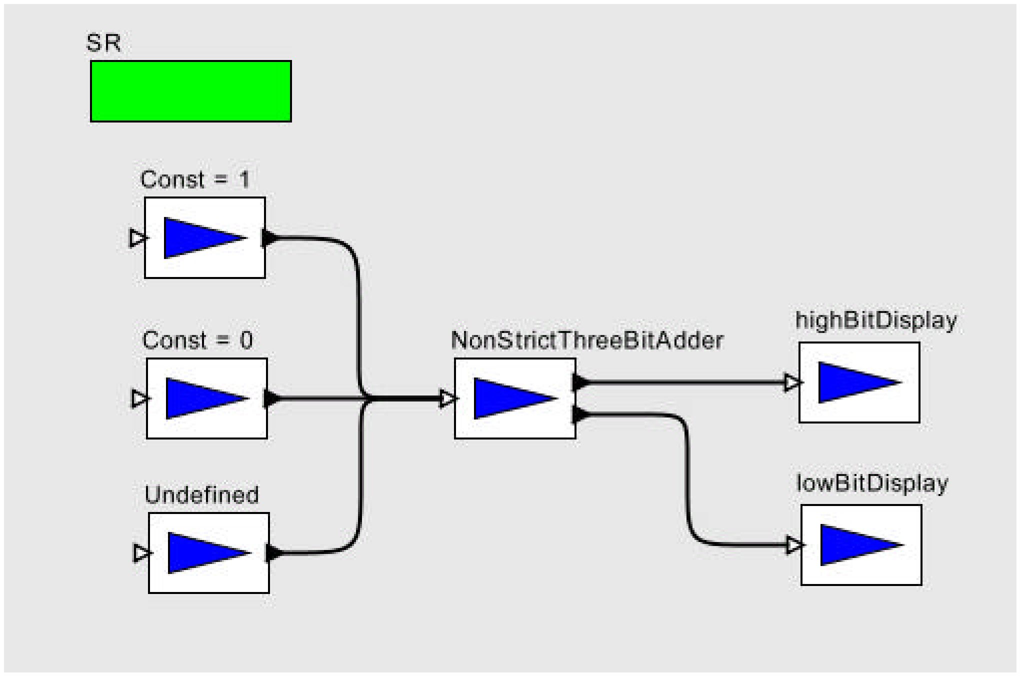
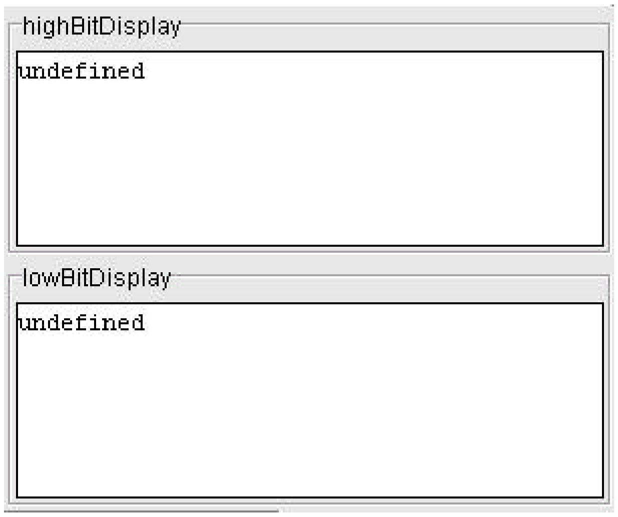


Figure 21. A model with a nonstrict three bit adder.

If this system is executed, the adder will attempt to produce partial results from the known inputs. If the undefined value was 0, the inputs would be 1, 0, and 0. The high bit output would be 0, and the low bit output would be 1. If the undefined value was 1, the inputs would be 1, 0, and 1. The high bit output would be 1, and the low bit output would be 0. Therefore, from the given known inputs, neither of the outputs can be definitively determined, so the adder produces no outputs.



```
undefined
```

```
undefined
```

Figure 22. The execution results of Figure 21.

Figure 23 is identical to Figure 21 with one exception: the value of both constants is 1.

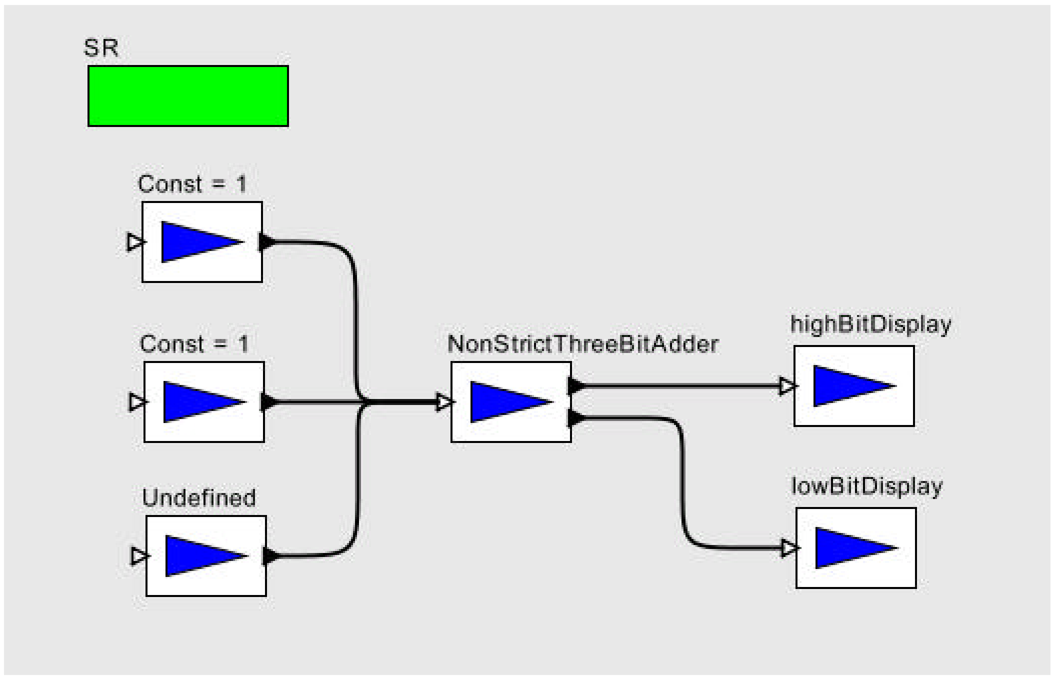
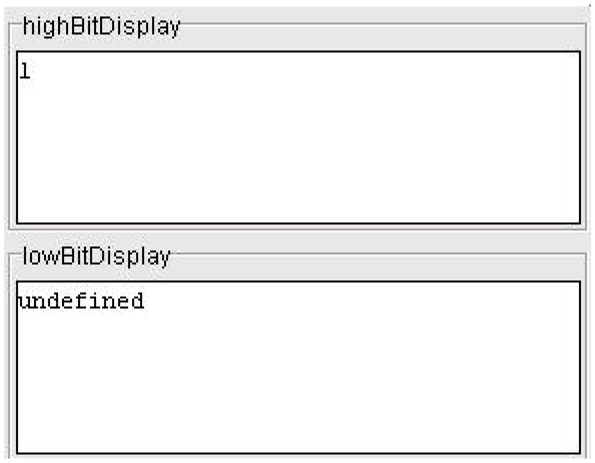


Figure 23. A second model with a nonstrict three bit adder.

If the undefined value was 0, the inputs would be 1, 1, and 0. The high bit output would be 1, and the low bit output would be 0. If the undefined value was 1, the inputs would be 1, 1, and 1. The high bit output would be 1, and the low bit output would be 1. Therefore, from these known inputs, the high bit output can be determined to be 1. The low bit, however, remains unknown.



```
1
```

```
undefined
```

Figure 24. The execution results of Figure 23.

In the case where all of the inputs to the adder were defined, SR would behave like many other domains, producing defined values for both outputs.

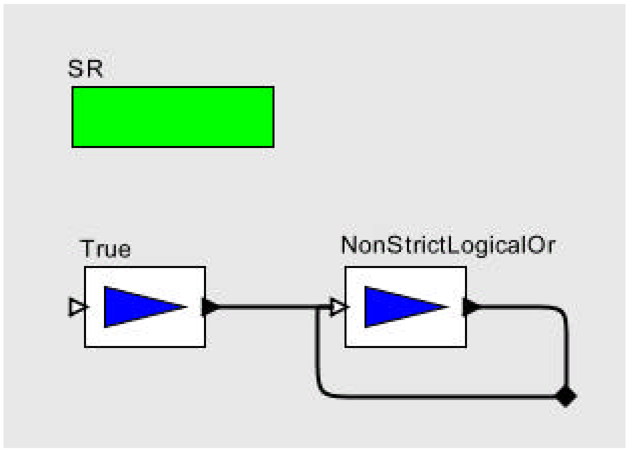


Figure 25. A cyclic graph.

Figure 25 shows a cyclic graph with an instantaneous feedback loop. When the execution of an instant begins, the NonStrictLogicalOr observes one input that is True and one input that is undefined. It can determine that the output should be True regardless of the value of the unknown input. This True value becomes the previously undefined input of the actor. All values are now known, and the model has reached a fixed-point. Note that the final results of the model are logically consistent.

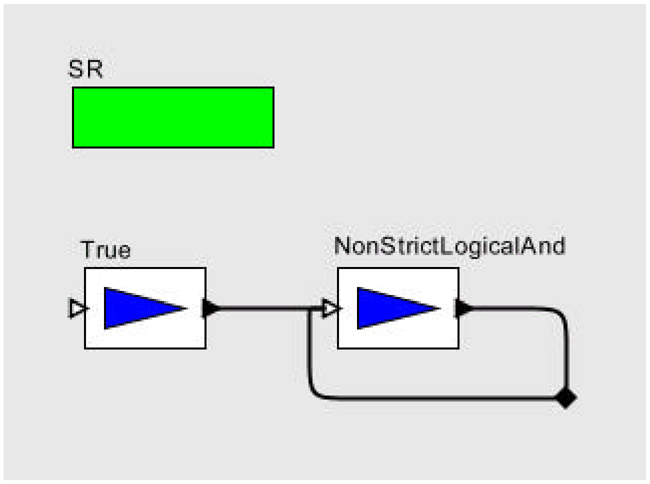


Figure 26. Another cyclic graph.

Figure 26 is very similar to the previous example, but in this case, the logic actor is a NonStrictLogicalAnd. When the execution of an instant begins, this actor observes one True input and one undefined input. If the unknown input was False, the actor would output False. If the unknown input was True, the actor would output True. Therefore, the output cannot be determined. No further values can be computed, so the iteration ends, and the output of the NonStrictLogicalAnd remains undefined.

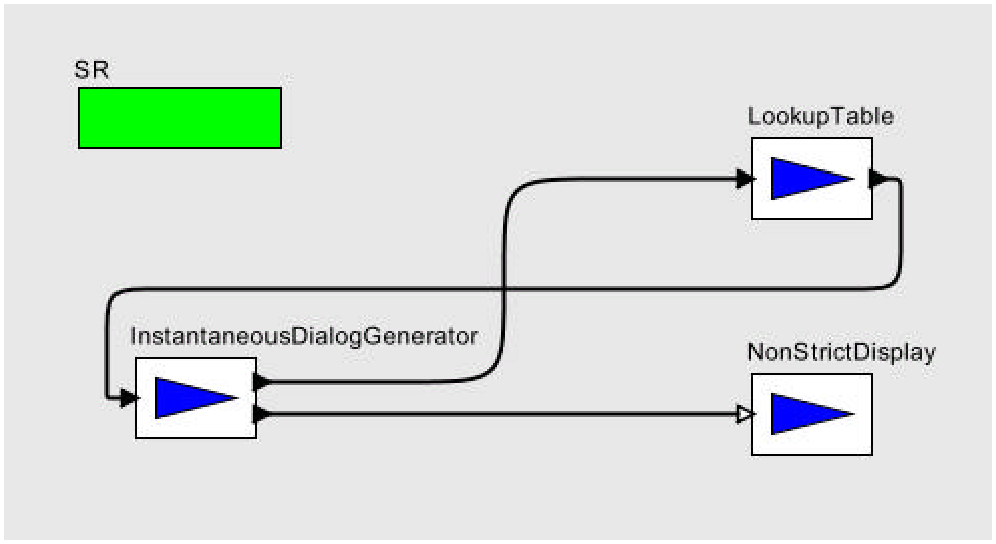
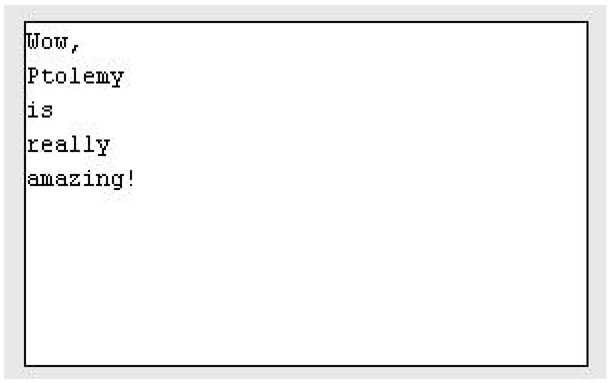


Figure 27. A model with instantaneous dialog.

In Figure 27, the specialized actor InstantaneousDialogGenerator both sends data to and receives data from a LookupTable. A LookupTable takes an integer index n as input and outputs the n^{th} element contained in the internal table as output. This particular LookupTable contains the following values:

index	element
0	"Wow,"
1	"Ptolemy"
2	"is"
3	"really"
4	"amazing!"

The InstantaneousDialogGenerator outputs incrementing integers (starting at 0), so the LookupTable will output its elements in sequential order. In the first iteration, the InstantaneousDialogGenerator outputs 0. The LookupTable then outputs "Wow,". When the InstantaneousDialogGenerator fires again, it will read this input and output it to the display actor. Since all of this happens in the course of a single iteration, the generator has engaged in an instantaneous dialog with the LookupTable. For the simplicity of this example, the InstantaneousDialogGenerator simply retransmits its input, but an actor could perform an arbitrary computation and output a different value. Figure 28 shows the results of five iterations of the model.



```
Wow,
Ptolemy
is
really
amazing!
```

Figure 28. The execution results of Figure 27.

4.2 SR embedded in SR

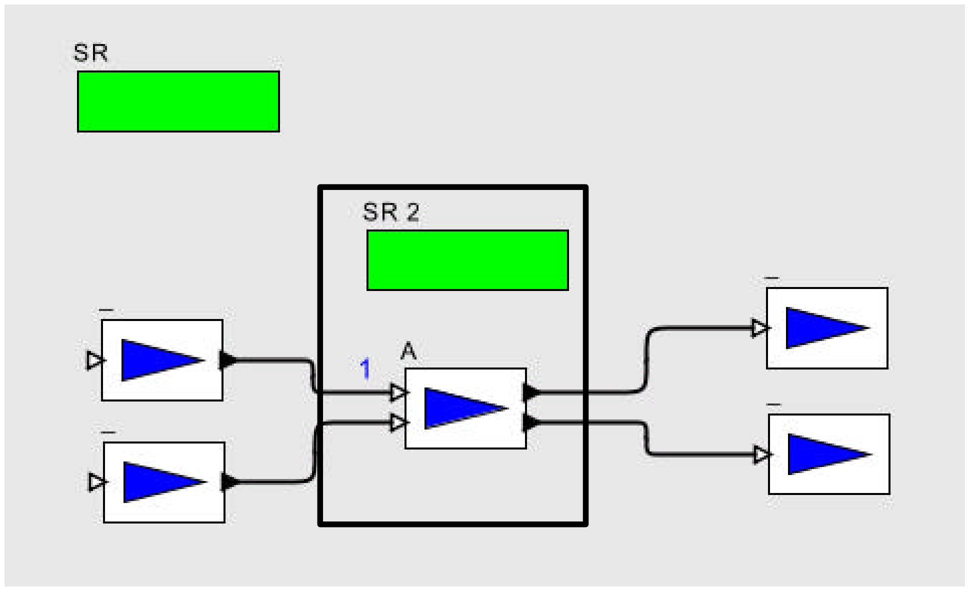


Figure 29. A hierarchical SR model.

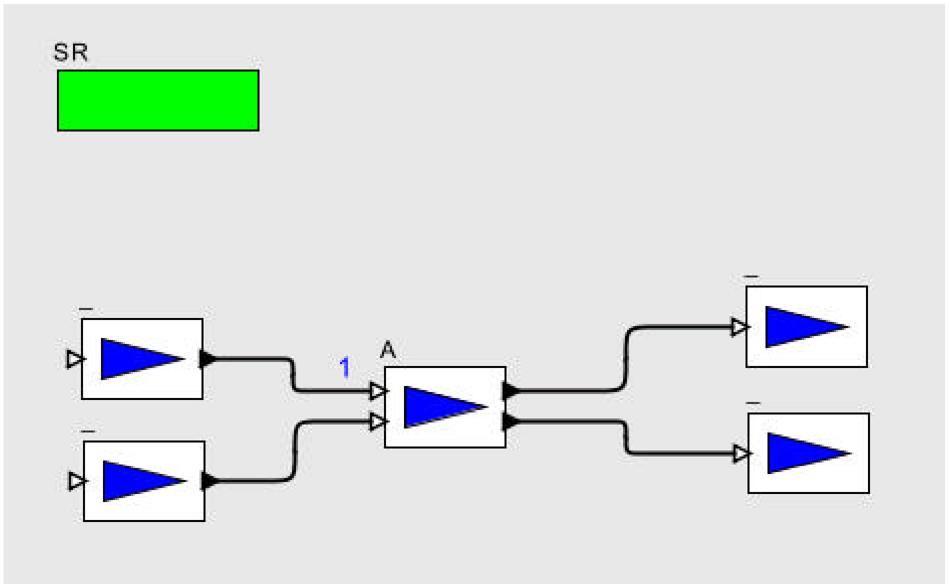


Figure 30. The same model, flattened.

An SR model embedded within an SR model essentially has the same behavior as the aggregate model with a single top-level SR director. However, a hierarchical SR in SR model will not behave exactly the same as the combined model if the inner composite actor is not treated as a nonstrict actor by the outer director. For example, in Figure 29, if the SR sub-model is considered to be strict and the value on channel 1 is unknown, the sub-model will not fire, and thus the nonstrict actor A will not fire.

If the sub-model in Figure 29 is treated as a nonstrict actor, the system is exactly equivalent to that in Figure 30. In fact, any hierarchical pure-SR model is equivalent to the same model with a single top-level SR director provided that all internal composite actors are treated as nonstrict.

4.3 SR embedded in other domains

When SR is embedded in another domain, the SR composite actor is fired by the outer director. The SR director inherits the current time from the outer director. The flexibility that hierarchical heterogeneity provides is illustrated by this model, which embeds SR in the Discrete-Time (DT) domain [8] to create a “Timed SR”:

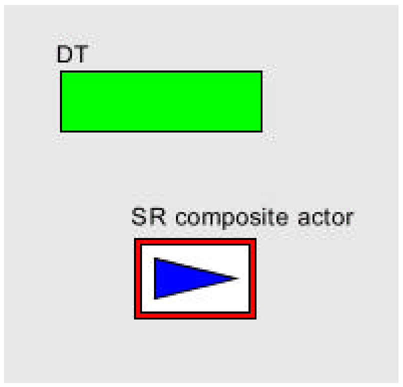


Figure 31. A DT model containing an SR composite actor.

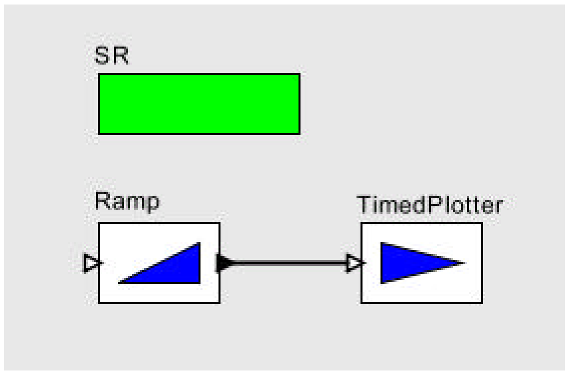


Figure 32. The contained SR model.

Executing this hierarchical model produces the results in Figure 33.

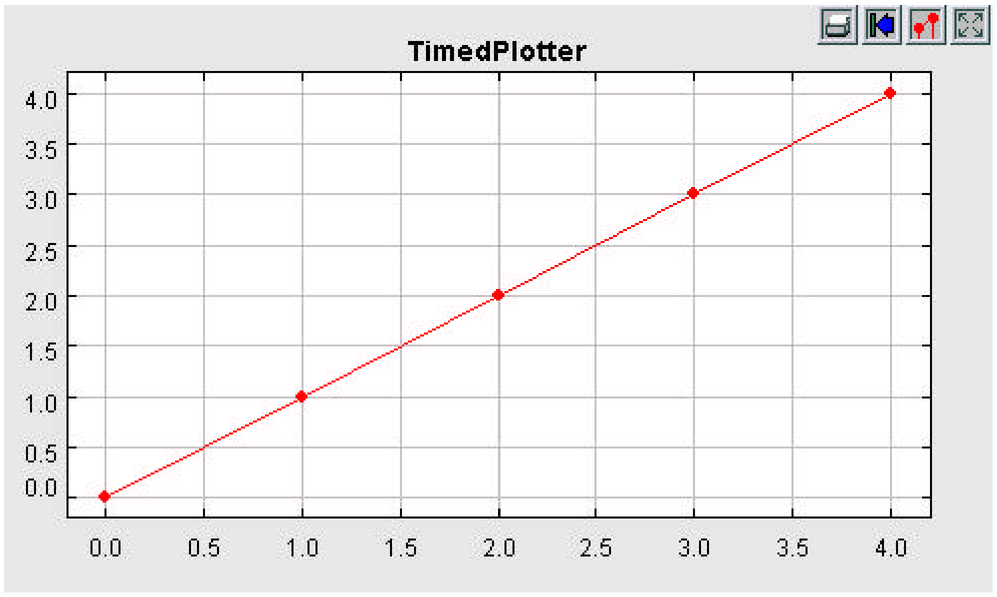


Figure 33. The execution results of Figure 31.

In any other domain, an embedded SR model is simply fired as an actor according to the semantics of the outer director.

4.4 Other domains embedded in SR

Embedding other domains inside SR can be very convenient. Since there are many existing actors and models that operate according to various models of computation, it is desirable to be able to use these components in SR without rearchitecting their functionality. These components can be inserted into an SR model in the Ptolemy II implementation.

It is fairly easy to incorporate foreign sub-models into an SR model. Hierarchical heterogeneity is handled in the manner described in the previous section. In most cases, the sub-model must be treated as a strict block, since the system may expect all of its inputs to be defined.

4.5 FSM embedded in SR

All nonstrict actors mentioned to this point have been specially programmed for specific purposes in SR. The Finite State Machines (FSM) domain [9] is a particularly interesting domain to embed in SR, because FSMs can be a rich source of customizable nonstrict actors.

In the FSM model of computation, the system is in one of a number of *states* at any given time. The system can change states via *transitions*, which are triggered by specified input conditions. Each transition may produce associated outputs.

In his Ph.D. dissertation, Bilung Lee [14] discusses FSM within SR. Ptolemy II was designed to allow actors to easily separate the production stage, in which outputs are produced, from the transition stage, in which state changes occur. An actor produces outputs in its fire() method but updates its state only in its postfire() method. Therefore, in a given iteration, an actor can be fired any number of times before postfire() is called exactly once to update its state.

If an SR model contains an FSM that is considered to be nonstrict, the FSM will be fired even if some inputs are unknown. If a particular transition includes unknown inputs in its input conditions, the transition simply does not occur, and no output is produced. If the inputs become known, and the FSM is fired later in the iteration, the outputs will then be produced.

Perhaps a more complete approach would allow a transition to occur if the result of the input conditions can be determined from the given known inputs, even if some of the inputs included in the conditions are unknown. My implementation does not handle such cases, as doing so would require substantial modifications to tool-generated expression parser code. However, these complex transitions can be expressed by separating the input conditions, creating multiple transitions.

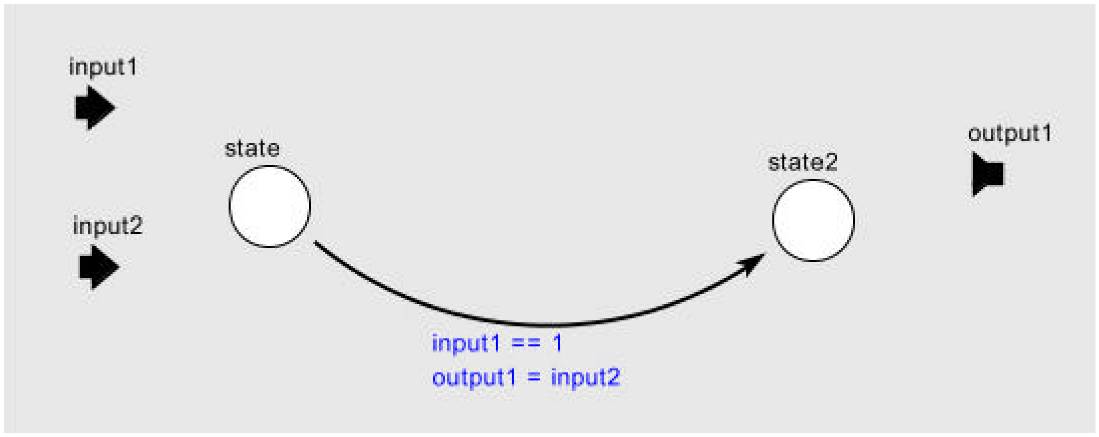


Figure 34. An FSM composite actor.

In the following example, an FSM is embedded inside an SR model. The FSM model is shown in Figure 34. The FSM composite actor has two inputs and one output. The FSM has a single transition: if the first input equals 1, the transition is enabled, and the value of the second input is produced on the output. We will consider four cases of inputs for this FSM embedded in SR.

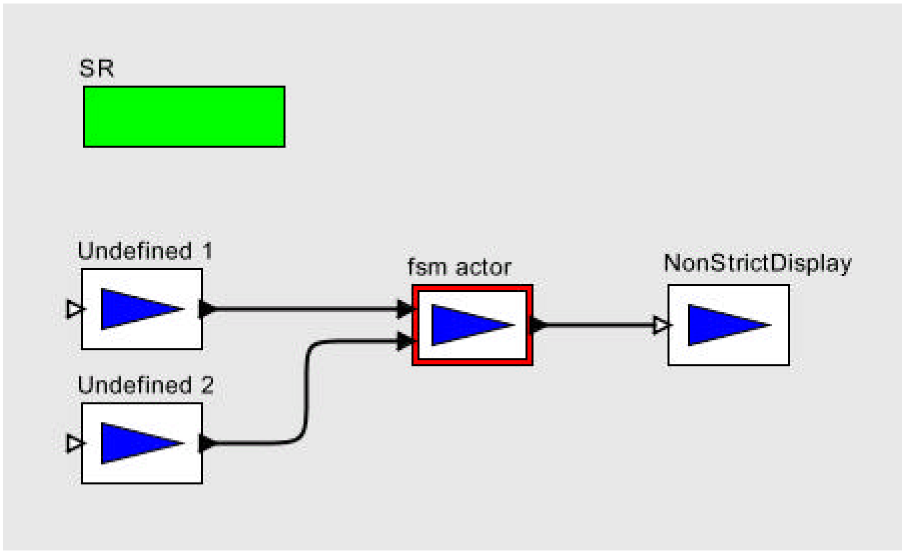


Figure 35. A model with two undefined inputs.

Case 1: Both inputs are undefined, as shown in Figure 35. In this case, the transition does not occur because the first input is unknown.

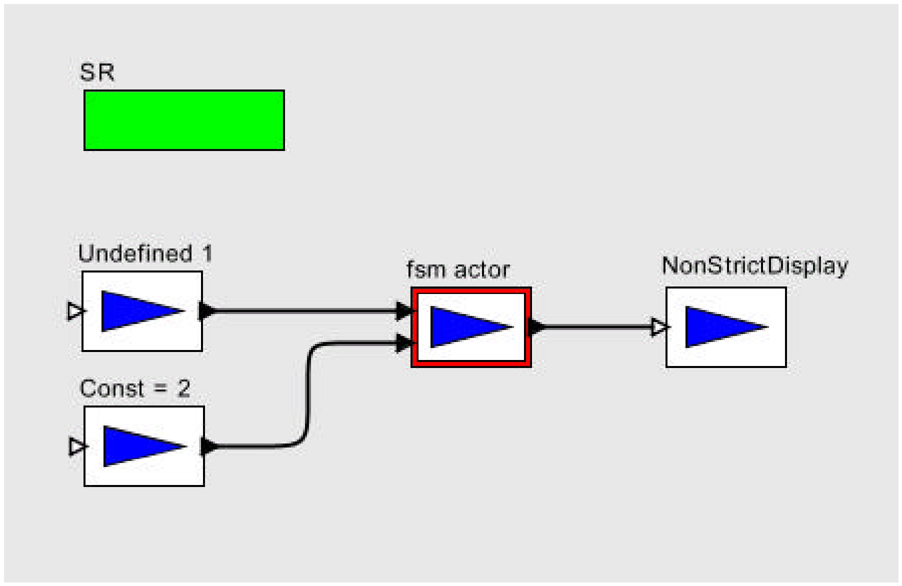


Figure 36. A model with one undefined input and one defined input.

Case 2: In Figure 36, the second input is defined, but the first is not. Again, the transition does not occur because the first input is unknown.

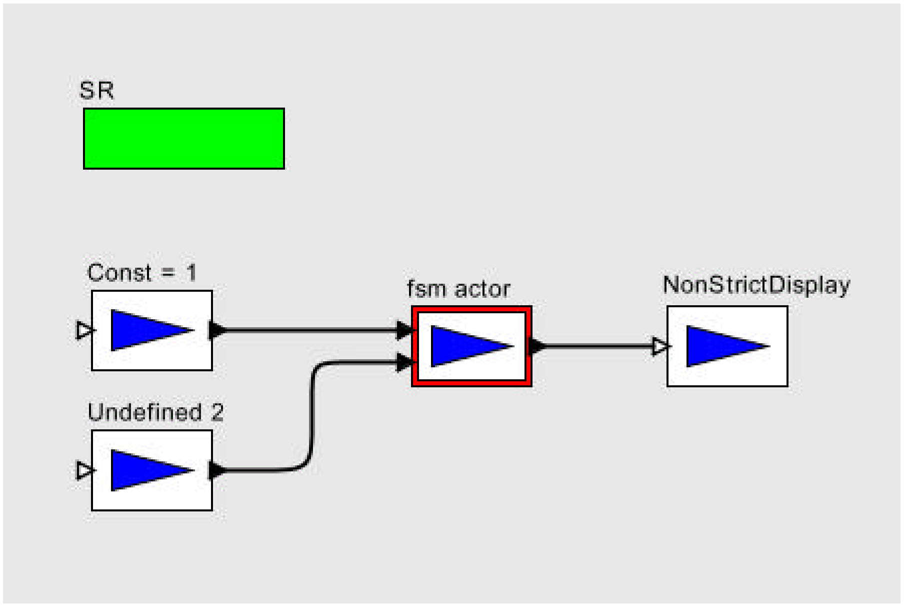


Figure 37. A model with one defined input and one undefined input.

Case 3: In Figure 37, the first input is defined, but the second is undefined. The input conditions to enable the transition are satisfied, but the value to output cannot be determined. Therefore no value will be produced at the output. However, the transition will be committed, and the FSM will change state upon the conclusion of the iteration.

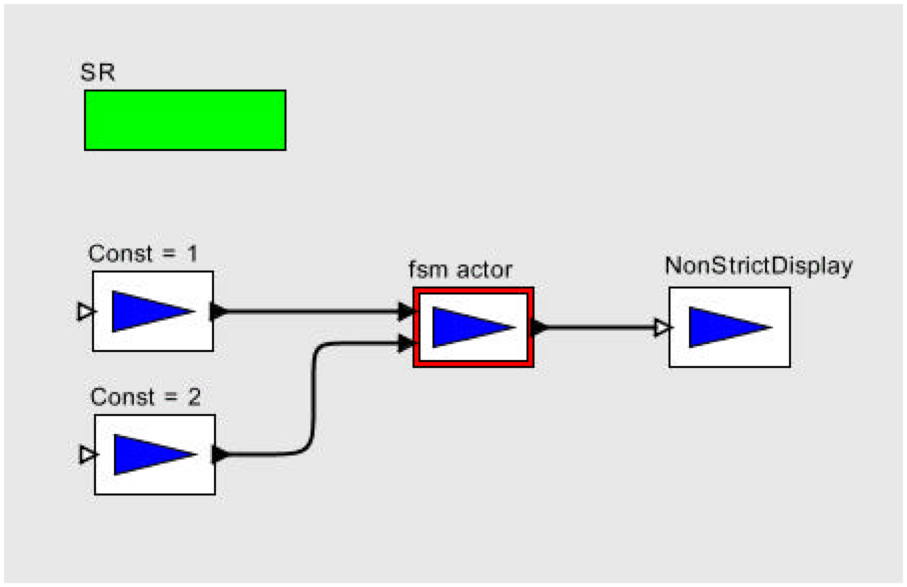


Figure 38. A model with two defined inputs.

Case 4: Both inputs are defined, as shown in Figure 38. The transition will occur, and the value of the second input, which is 2, will be produced as the output of the FSM. Again, the FSM will change state.

4.6 The Reflex Game

In this section, I present a heterogeneous model that embeds FSM within SR within DE. The system is an adaptation of the Reflex Game described by Berry and Gonthier [4]. Once the user starts the game, an event occurs after some randomly determined time interval. The user is expected to respond to the event as quickly as possible, and the system reports their reaction time.

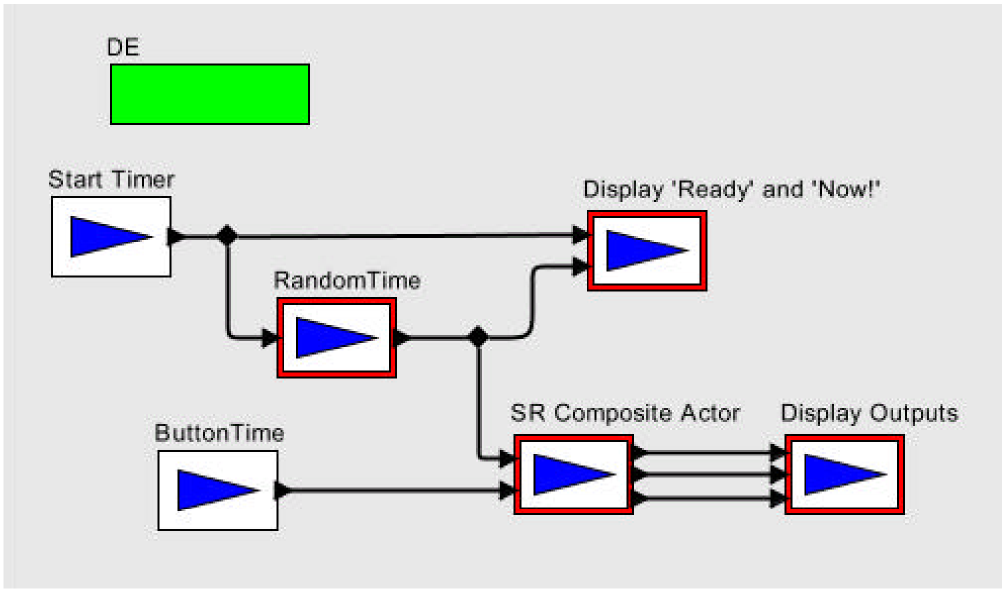


Figure 39. The Reflex Game.

Figure 39 shows the top-level DE model of the system. As we have seen before, a thick red border around an actor indicates that the actor contains a sub-model. We will run the system as a real-time model by setting the *synchronizeToRealTime* parameter of the DE Director to true. The *Start Timer* actor generates an event at time zero. This event triggers the firing of the *RandomTime* composite actor, which is shown in Figure 40.

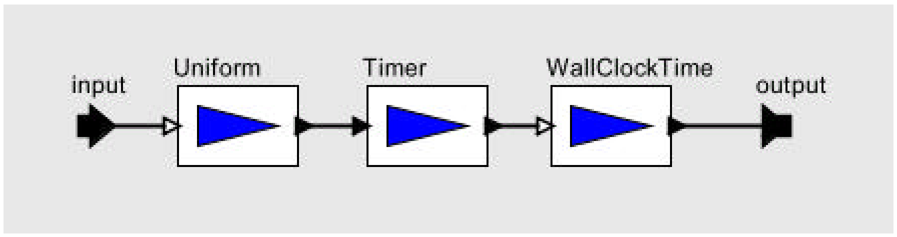


Figure 40. The RandomTime composite actor.

Since this sub-model has no director, it operates according to the semantics of the top-level DE director. When the *Uniform* actor receives a trigger input, it responds by producing a random number according to a uniform distribution. The distribution is set to have a range from 3.0 to 7.0. The *Timer* actor sets a timer to produce an event after the time period specified by this value (in seconds). Upon receiving this trigger, the WallClockTime actor produces a token that has the value of the time elapsed since the beginning of the execution of the model.

In the top-level model, the actor *Display 'Ready' and 'Now!'* will receive one event at time zero and one event at a random time between 3.0 and 7.0 seconds. Upon receiving the first event, this actor informs the user to be ready for the signal, and upon receiving the second event, the actor signals the user to respond.

The *ButtonTime* actor handles the response of the user by generating an event in response to a button click. Upon detecting a click, the actor requests immediate firing from the director. Upon firing, the *ButtonTime* actor outputs a token representing the elapsed time since the start of execution.

The *SR Composite Actor* receives two time values. One is the time of the randomly generated event, and the other is the time of the response by the user. Since the real-time capability of the DE Director is being used, each time value is received at the actual time of the corresponding event. The contents of the *SR Composite Actor* are shown in Figure 41.

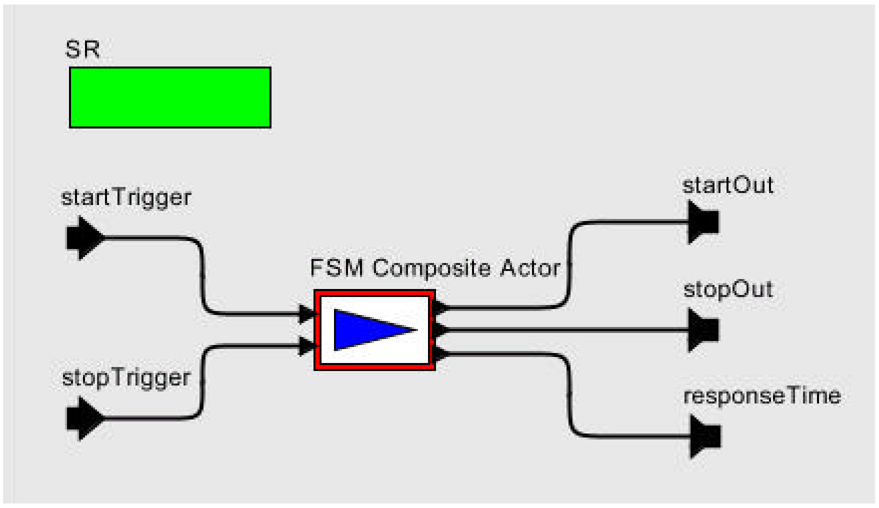


Figure 41. The SR Composite Actor.

This actor has two input ports and three output ports. In a given iteration, either of the inputs could be undefined. In fact, both inputs will be defined only if the two events occur at precisely the same time. The automaton inside the *FSM Composite Actor* is shown in Figure 42.

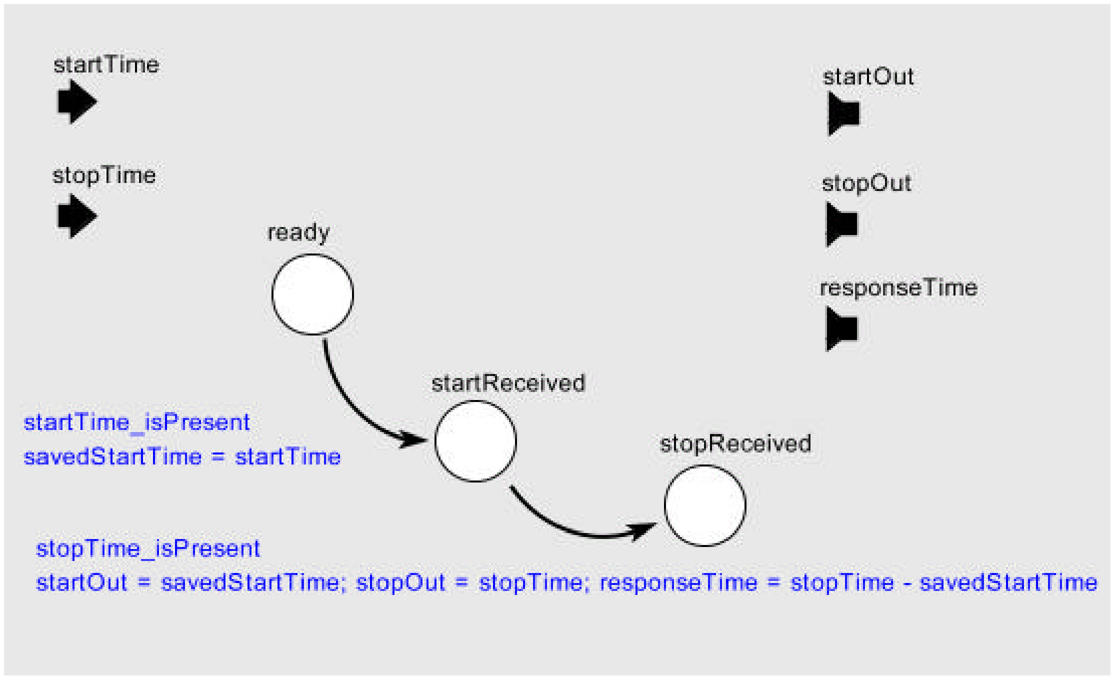

startTime
stopTime
startOut
stopOut
responseTime
ready
startReceived
stopReceived
startTime_isPresent
savedStartTime = startTime
stopTime_isPresent
startOut = savedStartTime; stopOut = stopTime; responseTime = stopTime - savedStartTime

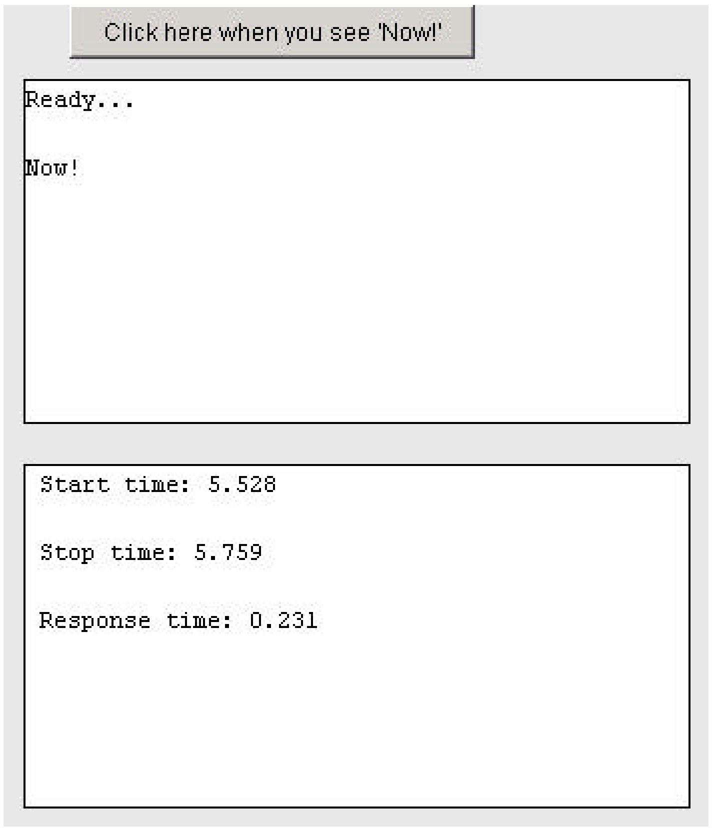
Figure 42. The FSM Composite Actor.

The FSM begins in the *ready* state. When the input *startTime* is present, corresponding to the randomly generated event, the FSM records the value of *startTime* in the parameter *savedStartTime* and transitions to the *startReceived* state. If this actor is fired when the *startTime* input is unknown, the transition will not occur since it is not possible to determine the presence of *startTime*.

Once this FSM is in the *startReceived* state, the presence of a *stopTime* input will cause three outputs to be produced. The original *startTime* that was recorded as the *savedStartTime* parameter will be produced on the port *startOut*. The value of the *stopTime* input will be output on the *stopOut* port. Also, the reaction time of the user will be calculated as the difference of these two times, and this value will be produced on the *responseTime* output.

Note that if the user were to click the button early, generating a *stopTime* input before a *startTime* input, the FSM would not make any transition from the *ready* state. Also, if this FSM were to receive a second *startTime* input while in the *startReceived* state, no transition would occur.

The *Display Outputs* composite actor in the top-level model presents the start time, stop time, and response time to the user. The results of an execution of the Reflex Game are shown in Figure 43.



```
Ready...

Now!
```

```
Start time: 5.528

Stop time: 5.759

Response time: 0.231
```

Figure 43. The execution results of the Reflex Game.

5. Conclusion

At first, SR may seem to be a very idealized model of computation, but we have shown that it can be used for practical applications. In fact, SR can be appropriate for modeling any system that reacts to its environment. Thus the Synchronous Reactive domain can be suitable for simulating user interfaces, objects that interact instantaneously, and any other real-world reactive system.

Fixed-point semantics are a natural way to model systems with zero-delay feedback. Deadlock does not occur, and determinacy can be ensured. Also, the fixed-point approach allows flexible component scheduling, so any model can be executed regardless of its configuration.

Using hierarchical heterogeneity in Ptolemy II, we can superimpose useful semantic properties of other domains onto SR. For example, we can introduce a notion of time simply by embedding SR inside any timed domain. It is also possible to describe different parts of an SR model using the models of computation that best describe their behavior.

Further work in this area might include specifying alternate ways to define nonstrict actors. Writing Java code to define these actors is inconvenient, since actors must be custom-written for most applications. Creating finite state machines that are nonstrict is significantly easier, but less expressive. Perhaps some innovative scheme could provide the advantages of each.

6. References

[1] T. Amagbegnon, L. Besnard, P. Le Guernic, “Implementation of the Data-flow Synchronous Language Signal,” Programming Languages: Design and Implementation, ACM, 163-173, 1995.
ftp://ftp.irisa.fr/local/signal/publis/articles/PLDI-95:compil.ps.gz

[2] Albert Benveniste and Gérard Berry, “The Synchronous Approach to Reactive and Real-Time Systems,” Proceedings of the IEEE, Vol. 79, No. 9, 1270-1282, September 1991.

[3] Gérard Berry, “The Foundations of Esterel,” Proof, Language, and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling, and M. Tofte, editors, MIT Press, Foundations of Computing Series, 2000.
ftp://ftp.esterel.org/esterel/pub/papers/foundations.[pdf/ps]

[4] Gérard Berry and Georges Gonthier, “The Esterel Synchronous Programming Language: Design, Semantics, Implementation,” Science of Computer Programming, Vol. 19, No. 2, 87-152, November 1992.
ftp://ftp-sop.inria.fr/meije/esterel/papers/BerryGonthierSCP.[pdf/ps]

[5] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” International Journal of Computer Simulation, special issue on “Simulation Software Development,” Vol. 4, 155-182, April 1994.
http://ptolemy.eecs.berkeley.edu/publications/papers/94/JEurSim/

[6] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong, “Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java,” Technical Memorandum UCB/ERL M01/12, EECS, University of California, Berkeley, March 2001.
http://ptolemy.eecs.berkeley.edu/publications/papers/01/HMAD/

[7] Stephen A. Edwards, “The Specification and Execution of Heterogeneous Synchronous Reactive Systems,” Ph.D. thesis, Technical Memorandum UCB/ERL M97/31, EECS, University of California, Berkeley, May 1997.
http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/

[8] Chamberlain Fong, “Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II,” M.S. Report, Technical Memorandum UCB/ERL 01/9, EECS, University of California, Berkeley, January 2001.
http://ptolemy.eecs.berkeley.edu/publications/papers/00/dt/

[9] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems, Vol. 18, No. 6, June 1999.
http://ptolemy.eecs.berkeley.edu/publications/papers/99/starcharts

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Dataflow Programming Language Lustre," Proceedings of the IEEE, Vol. 79, No. 9, 1305-1320, September 1991.
http://www-verimag.imag.fr/PEOPLE/Nicolas.Halbwachs/lustre:ieee.html

[11] Edward A. Lee, "Computing for Embedded Systems," IEEE Instrumentation and Measurement Technology Conference, Budapest, Hungary, May 2001.
http://ptolemy.eecs.berkeley.edu/publications/papers/01/embeddedBudapest/

[12] Edward A. Lee, "Synchronous Data Flow," Proceedings of the IEEE, Vol. 75, No. 9, 1235-1245, September 1987.

[13] Edward A. Lee and Alberto Sangiovanni-Vincentelli, " A Framework for Comparing Models of Computation," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, No. 12, December 1998.
http://ptolemy.eecs.berkeley.edu/publications/papers/98/framework/

[14] Bilung Lee, "Specification and Design of Reactive Systems," Ph.D. dissertation, Technical Memorandum UCB/ERL M00/29, EECS, University of California, Berkeley, May 2000.
http://ptolemy.eecs.berkeley.edu/publications/papers/00/reactiveSystems/

[15] F. Maraninchi, "The Argos Language: Graphical Representation of Automata and Description of Reactive Systems," Proceedings of the IEEE Workshop on Visual Languages, Kobe, Japan, October 1991.
http://www-verimag.imag.fr/~maraninx/IEEE-VISUAL91.html

[16] Lukito Muliadi, "Discrete Event Modeling in Ptolemy II," M.S. Report, EECS, University of California, Berkeley, May 1999.
http://ptolemy.eecs.berkeley.edu/publications/papers/99/deModeling/