Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II

by Chamberlain Fong

Research Project

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 M01/9
December 2000

Abstract

The Discrete Time (DT) domain in Ptolemy II is a timed extension of the Synchronous Dataflow (SDF) domain. Although not completely backward compatible with SDF, DT keeps most of the desirable properties of SDF like static scheduling, regular/periodic execution, bounded memory usage, and a guarantee that deadlock will never occur. In addition, DT has some desirable temporal properties such as uniformly-timed token flow and causality. This paper will present the semantics and implementation of the DT domain in Ptolemy II. This paper will also present the DT domain working with other domains in Ptolemy II. In particular, it will present applications of DT working with the Graphics (GR) domain for 3D animated simulations.

Acknowledgements

I would like to thank my research advisor, Professor Edward A. Lee, for introducing me to this research area. I appreciate all the support and guidance that he has given me. I really enjoyed my meetings with him where we brainstormed through the different twists and pitfalls of research in dataflow computing.

I would also like to thank Professor Alberto Sangiovanni-Vincentelli, the second reader of this report. His EE249 class gave me solid background on the different models of computations.

I would also like to thank all my colleagues in the Ptolemy group. I really appreciate the help given to me by Brian Vogel, Steve Neuendorffer, Jie Liu, and Xiaojun Liu while I was trying to learn to use the Ptolemy II system.

Lastly, I would like to thank my parents for their continuous support and encouragement. This report is dedicated to them.

Table of Contents

List of Figures

1. Introduction

1.1 Dataflow Computing

We are all familiar with the Von-Neumann/stored-program execution model for computing. However, there are several viable alternatives to the Von-Neumann execution model. For example, there are computers with execution models based on neural networks, analog electronics, and DNA molecules. Dataflow is another example of an alternative model for computing. Dataflow consists of computing blocks that are connected in a directed graph. The computing blocks process data from their inputs and produce data on their outputs. Aside from processing I/O and performing computation, the blocks also communicate with each other through their connections in the underlying dataflow graph. Dataflow is commonly used in block diagram programming environments such as our software system – Ptolemy II.

1.2 Ptolemy II

Ptolemy II [4] is a software infrastructure currently under active development in the EECS department of the University of California, Berkeley. Ptolemy II can be thought as a visual programming language that supports hierarchy. Its primarily focus is on the design and simulation of complex heterogeneous systems. These systems are complex and heterogeneous in the sense that they mix widely different operations, such as signal processing, feedback control, 3D visualization, and user interfaces. Users of the Ptolemy II software design their systems through components that encapsulate computing capability. Ptolemy II interprets the execution and interaction semantics of its components through what we call *domains*. Depending on the *domain*, the components can represent subroutines, threads, processes, hardware IP, or even mechanical parts. Each domain has a well-defined *model of computation* that specifies the formal semantics of component behavior in the given domain. Figure 1 shows a screenshot of Ptolemy II. In particular, it shows *Vergil*, the front-end graphical user-interface of Ptolemy II. We will be using a lot of Vergil screenshots in the figures used in this paper. This will help illustrate our concepts more clearly.

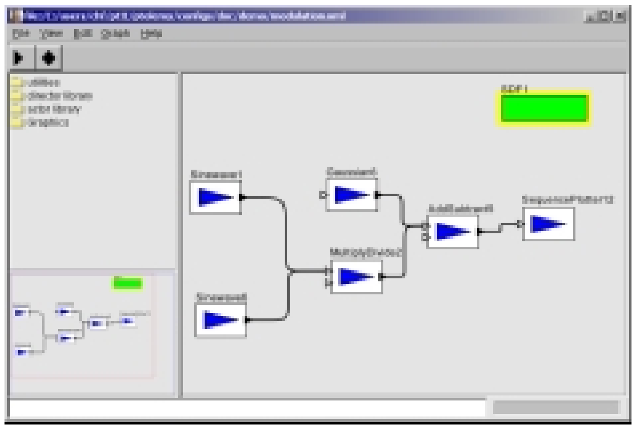


Figure 1: Screenshot of Vergil

The components in Ptolemy II are called *actors*. The actors are connected in a directed graph topology by links called *relations*. Each actor has input and output *ports* for which they communicate with other actors. The actors communicate with each other by passing messages called *tokens*. The actors produce and consume tokens during execution (also known as *firing*) of the actor. In order to facilitate the firing of the actors, each domain in Ptolemy II has its own director and receiver class. The *director* coordinates the scheduling and execution of actors within the domain. The *receivers* contain the necessary buffers for token flow. In some domains, the director also keeps track of time and concurrency of the actors.

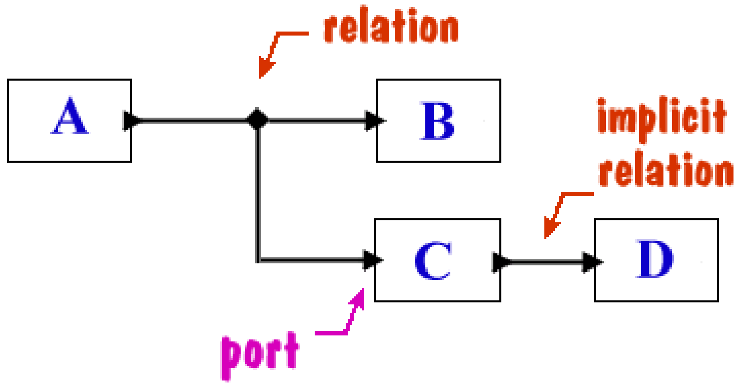


Figure 2. A Ptolemy II component-based graph. The boxes labeled A, B, C, and D are the actors. The edges that link the actors are called implicit relations. The edges with diamond-shaped splitters are called explicit relations. The arrows where the relations/links come in and out of the actors are called ports

1.3 Models of Computation

Models of computation [8] form the mathematical underpinning of each domain in Ptolemy II. More specifically, a model of computation provides an abstract set of rules in which the actors in the given domain behave. These rules include: the order (also known as *schedule*) in which the actors are fired; the progression of time; and the amount of buffer memory in each of the actors' ports. Instead of explaining these concepts using elaborate examples, we will provide some important questions that motivate the use of models of computation. These questions will be answered in the latter sections of this paper.

- There are several different ways to order the sequence in which the actors are fired. How do I know which one is valid? Is the schedule computed at compile-time or at run-time?
- How many tokens can an actor consume and produce at each firing? Should an actor stall when there are no tokens available when it fires?
- How much buffer memory should be allocated for each receiving port of the actors? What happens when more tokens arrive and the receiving buffer is full?
- What is the meaning of time? What is the minimum time-slice in the progression of time? Does computation explicitly take time?

An important principle of Ptolemy II is that the choice of models of computation strongly affects the quality of a system design. A model of computation is analogous to the “laws of physics” in the sense that it governs the interaction of actors in the model. The Ptolemy II software provides a rich variety of models of computation that deal with concurrency and time in different ways. Here is a rundown of the models of computations currently provided in Ptolemy II:

CSP	-	communicating sequential processes
CT	-	continuous time
DDE	-	distributed discrete event
DE	-	discrete event
DT	-	discrete time (the main subject of this report)
FSM	-	finite state machines
PN	-	process networks
SDF	-	synchronous dataflow

As an aside, we would like to mention that some actors in Ptolemy II are *domain-specific*; i.e. they can only work with under certain models of computations. The ‘integrator’ actor in the CT domain is an example of this. In contrast, there are *domain-polymorphic* actors in Ptolemy II. These actors work under all models of computations. The ‘Ramp’ and ‘Scale’ actor are examples of domain-polymorphic actors. Please refer to the Ptolemy II documentation [4] for more details on these actors.

1.4 Heterogeneous Hierarchies

One of the most important features of Ptolemy II is that different models of computation can be hierarchical composed to create complex and elaborate systems. In order to support hierarchy, Ptolemy II has two types of actors – *composite actors* and *atomic actors*. Composite actors are actors that contain other actors inside them, much like a directory contains files. On the other hand, atomic actors cannot contain any actors inside. For example, figure 3 shows a graph of simple hierarchical model in Ptolemy II. The model has three actors: A, B, and C on the top level. Actor B is a composite actor that contains two actors inside, D and E. On the other hand; actors A, C, D, and E in the model are atomic actors. The nesting of actors inside actors is the basis for hierarchy in Ptolemy II.

Composite actors can also contain directors. When a composite actor contains a director, it is said to be an *opaque composite actor*. When a composite actor does not contain a director, it is said to be a *transparent composite actor*. Opaque composite actors are much more common in Ptolemy II models because they form the basis for mixing different models of computations. The director that directs an opaque composite actor is called the *outside* or *executive director*. The director contained inside an opaque composite actor is called the *inside director*.

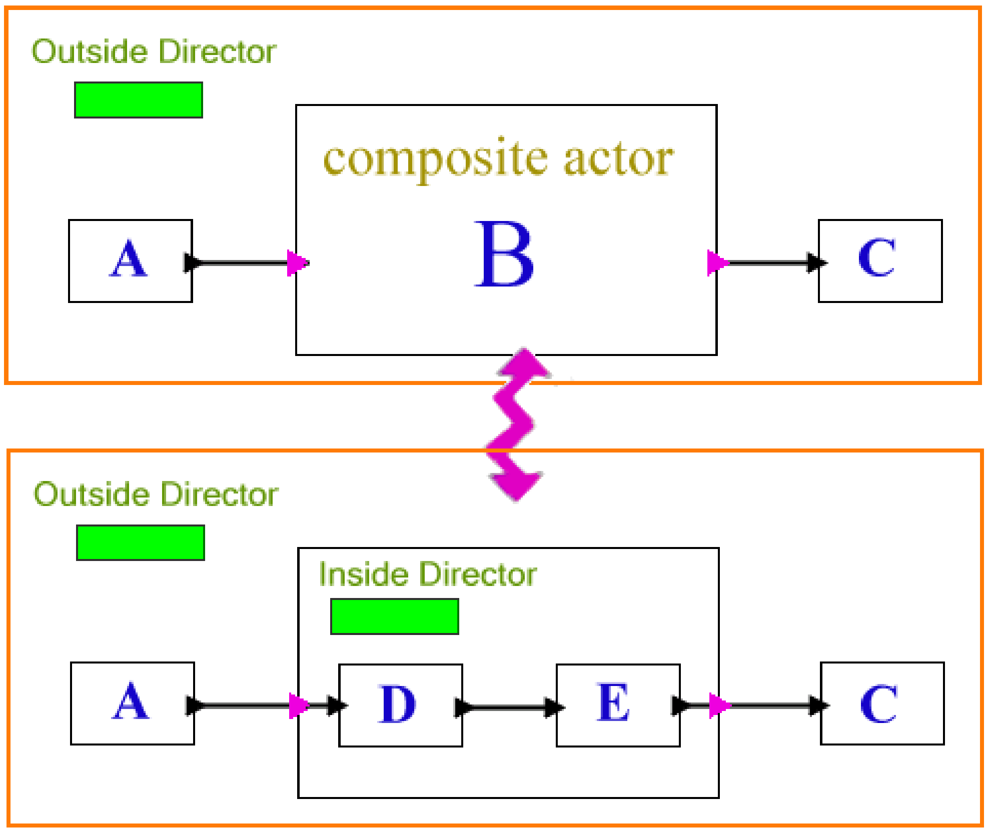


Figure 3. Outside and Inside Directors for opaque hierarchies

2. Discrete Time (DT) Domain

2.1 Overview

DT was first-and-foremost designed to be a timed-extension of SDF. SDF is a standard and well-understood model of computation in Ptolemy Classic and Ptolemy II. However, SDF lacks semantics for time; and hence doesn't have some desirable temporal interaction semantics when hierarchically linked with other timed models of computation like Discrete Event (DE) and Continuous Time (CT). The prospect of SDF working with other models of computations is the main motivation for extending it into a timed domain. As an example, consider the simple Ptolemy II diagrams shown below. We have a simple model of two actors: a ramp and a timed plotter. The diagram on the left is under an SDF director; and the diagram on the right is under a DT director. The results of the execution are shown as plots below the diagrams. The SDF diagram produces a plot that is fixed at time = 0.0 seconds. All results of the SDF execution conceptually occur at time = 0.0 seconds. This is largely due to the fact that SDF is an untimed domain; and that connecting a ramp with a timed plotter under SDF does not have much meaning. On the other hand, the DT plot shows a more reasonable result where the ramp output values increase over time.

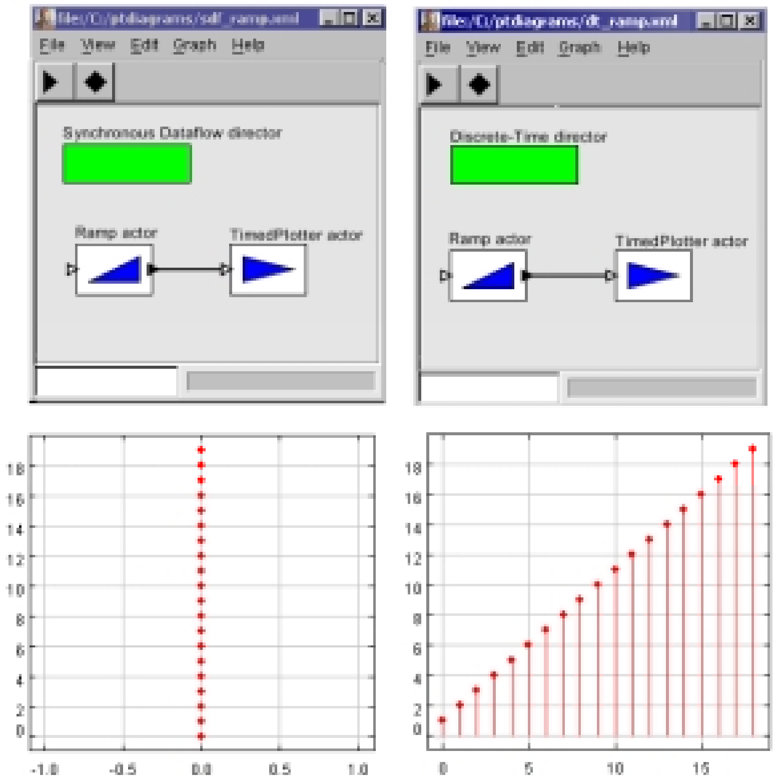

file:/C:/ptdiagrams/sdf_ramp.xml
File View Edit Graph Help
Synchronous Dataflow director
Ramp actor
TimedPlotter actor
file:/C:/ptdiagrams/dt_ramp.xml
File View Edit Graph Help
Discrete-Time director
Ramp actor
TimedPlotter actor
18
16
14
12
10
8
6
4
2
0
-1.0
-0.5
0.0
0.5
1.0
18
16
14
12
10
8
6
4
2
0
0
5
10
15

Figure 4. A comparison of SDF and DT

There is another view for which we can present DT. Ptolemy II has a rich software infrastructure for analyzing and modeling systems and signals. Natural and man-made systems are usually described using continuous-time and discrete-time models. Continuous-time systems are governed by differential equations. Discrete-time systems are usually sampled versions of continuous-time systems; and are governed by difference equations. Mathematical tools like the Laplace and Fourier transform are used to analyze continuous-time systems. In contrast, Z-transforms and discrete-time Fourier transforms are used to analyze discrete-time systems. Ptolemy II has a fairly mature infrastructure for continuous-time modeling, which is implemented in the CT domain. It turns out that Ptolemy II also has the necessary infrastructure for discrete-time modeling, too. SDF is excellent in handling sampled systems and difference equations. However, there are bits and pieces missing in SDF for discrete-time modeling. To be more specific, there is no concept of the progression of time in the SDF domain; hence we extend it to the DT domain. Figure 5 shows the basic difference between continuous-time and discrete-time models in Ptolemy II.

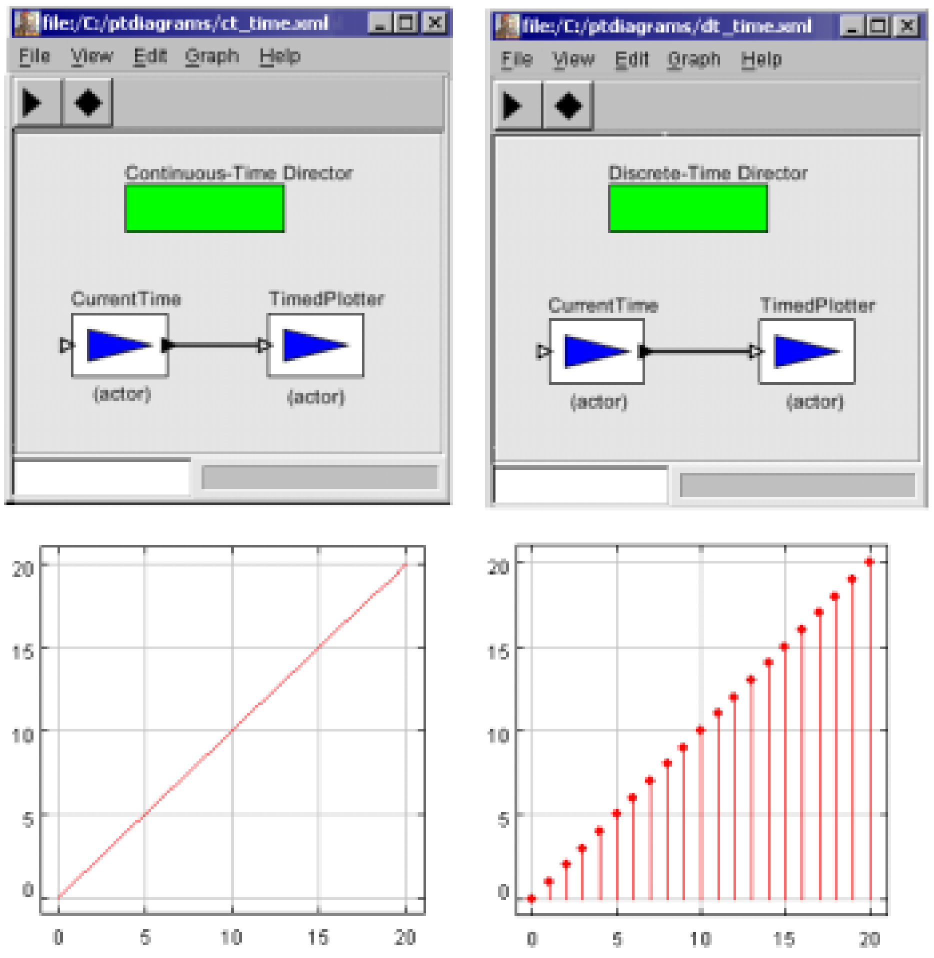


Figure 5. A comparison of CT and DT

Although the core of this paper is about the theory and implementation of the DT domain in Ptolemy II, we will provide simple applications at the end of this paper. In particular, we will focus our applications on simulation and computer graphics animation. Of course, computer graphics is not the only application of DT; we just happen to have a preference for it.

We have built a new experimental domain called GR (short for graphics) to handle computer graphics rendering and geometry management in Ptolemy II. The GR domain is an untimed domain, so it is useful to couple it with DT to produce animated simulations. We must now admit that continuous-time models might be more appropriate in computer graphics applications, especially those that simulate physical phenomena such as a swinging pendulum. However, the computer that displays the simulation still has to do the animation in discrete-time. We normally view computer-generated animations at about 15-30 frames per second in discrete time.

2.2 Synchronous Dataflow (SDF)

The Synchronous Dataflow (SDF) domain [9][10] was originally devised for component-based design of multi-rate signal processing algorithms. Figure 6 shows an example of component-based design under SDF. In particular, it shows an SDF model for simulating the 'eye diagram' in communication systems. SDF provides an abstract and rigorous dataflow specification for executing actors. Moreover, SDF models have several desirable properties that are proven to hold. These properties include: relatively simple code-generation, and a guaranteed upper bound on the buffer memory required for each of the ports. Later work has generalized SDF into more powerful abstract models of computation such as Boolean Dataflow (BDF), Dynamic Dataflow (DDF), and Process Networks (PN) [2][11].

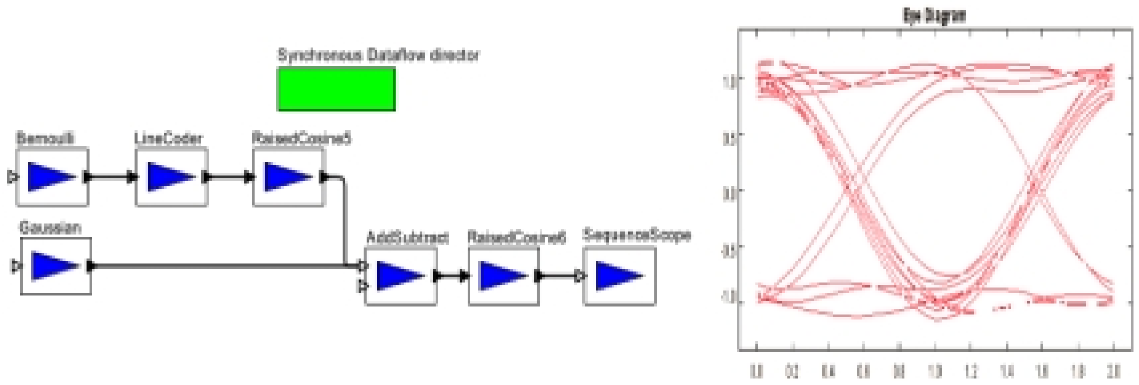


Figure 6. An SDF model (left) for simulating the eye-diagram (right) in communication systems.

Under the SDF domain, the execution order of the actors is statically determined prior to execution. In other words, the firing sequence (also known as *schedule*) of the actors can be pre-calculated and cached during compile-time or load-time. This schedule is then periodically repeated during simulation. Every periodic execution of a schedule is called an *iteration*. There is very little overhead on the simulation kernel during execution. This means that most of the time associated with simulation can be devoted to the actors performing their computations. This also means that the kernel devotes very little time to figuring out which component should execute next. This is a very desirable property of SDF that DT inherits.

Before we proceed, it is convenient to change gears and jump to a more abstract setting. Instead of dealing with real SDF models that are applied to real-world circuits and systems, we will deal with abstract *SDF graphs*. It is more convenient to study the properties of SDF and DT using abstract mathematical concepts. An SDF graph is a connected directed graph with positive integer labels (implicitly or explicitly) on each end of every edge. Figure 7 shows an example of an SDF graph. The vertices labeled with capital letters are the actors.

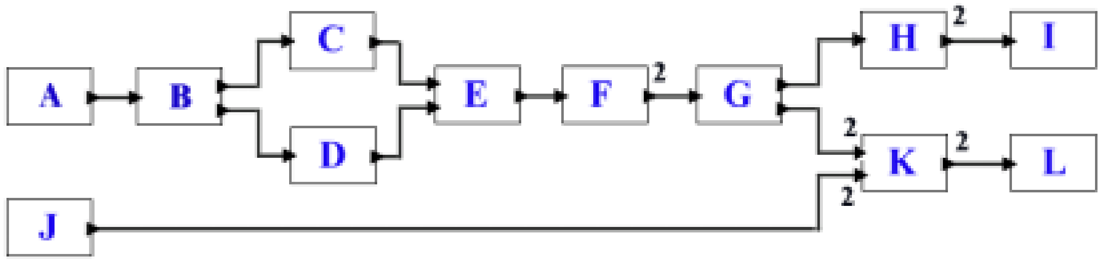


Figure 7. An abstract SDF graph

2.3 Port Rates and Actor Repetitions and Schedules

By definition, every SDF actor has fixed consumption and production rates on its ports. That is, for every firing of an actor, a fixed number of tokens are consumed and produced. This property makes it possible to statically schedule SDF graphs for execution. We will talk more about scheduling in the latter part of this section, but first, let us cover some basic definitions and concepts in SDF.

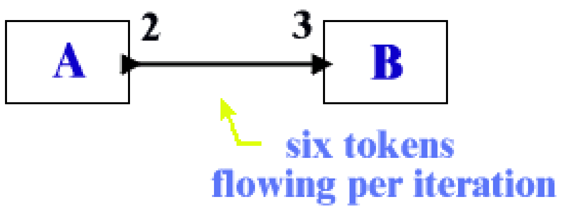


Figure 8. A simple SDF graph with possible schedule (3A)(2B).

Consider the simple SDF graph shown in figure 8. Actor A produces 2 tokens at every firing. Actor B consumes 3 tokens at every firing. In SDF jargon, we say that the output port of actor A has a *token production rate* of 2. Similarly, we say that the input port of actor B has a *token consumption rate* of 3. It is important that all tokens produced by actor A are consumed by actor B. We can think of this as the principle of conservation of tokens. All SDF graphs conserve tokens. We want to repeatedly fire actors A and B a certain number of times so that no tokens are left over afterwards. Let us denote the number of firings per iteration of actor A as r_A and the number of firings of actor B as r_B. If we apply the principle of conservation of tokens, we get the Diophantine equation: (also called an SDF *balance equation*)

$$2\ r_A = 3\ r_B$$

The smallest positive integer solution to this equation is r_A=3 and r_B=2. These values are called the *actor repetitions* of actors A and B, respectively. More complicated SDF graphs have multiple balance equations. In general, there is a balance equation for every edge of an SDF graph. Getting back to the diagram, we see that firing actor A thrice and firing actor B twice causes six tokens to flow from actor A to actor B with no leftover tokens on the edge. We shall denote the number of tokens flowing per iteration on an edge as its *token flow rate*. In the SDF graph shown below, the token flow rates of edges AB, BC, and CD are 36, 12, and 24 respectively. The token flow rate of an edge is an important property that we will refer to again later when we discuss DT concepts. Incidentally, the keen-eyed reader might observe that the output port of actor B and the input port of actor C do not have labeled rates. The convention for this paper is that ports without labeled rates have an implicit rate of value one. Such ports are also known as *homogeneous ports*. Actors that only have homogeneous ports are called *homogeneous actors*.

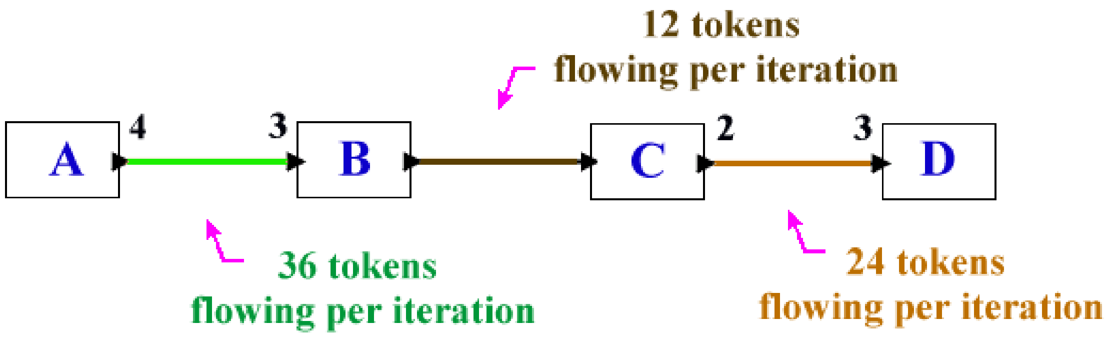


Figure 9 Another SDF graph (Note the convention used throughout this paper – ports without labeled rates have an implicit port rate of value one)

Once the balance equations of an SDF graph are solved, the token flow rates of an edge can be calculated using the following equation:

token flow rate = actor repetitions * port rate

There are actually two ways to view the previous equation. Since every edge connects a source and destination actor in an SDF graph, we can interpret the previous equation either from the source actor's viewpoint or from the destination actor's viewpoint. From the source actor's viewpoint, the port rate is its production rate. From the destination actors' viewpoint, the port rate is its consumption rate.

The balance equations of an SDF graph play a key role in the scheduling of the SDF graph. Those SDF graphs that do not have solutions to their balance equations are said to be not *schedulable*. Recall that an SDF schedule is a sequential ordering of the actors for execution. Also recall that the director periodically fires the actors in an SDF graph according to the schedule sequence. For example, the SDF graph shown in figure 8 can have a possible schedule of (3A)(2B). This notation is shorthand for a schedule that fires actor A thrice consecutively and then fires actor B twice consecutively. We must now note that SDF schedules are not necessarily unique. The given SDF graph can have a different schedule: A(2AB). In ordinary words, this schedule means fire actor A once and then fire actor A and B in sequence twice. Another important concept worth noting is *determinism*. Determinism in an SDF graph means that the end result of an iteration over the graph should be the same no matter what schedule is used.

There is a fair amount of literature on SDF scheduling. For our purposes, we just need to know what SDF schedules are and how to read SDF scheduling notation. We will not go into the details of SDF scheduling algorithms and SDF graph theorems. The enthusiastic reader is motivated to read the original SDF paper [10]. The paper gives more details on the formal semantics and scheduling of SDF graphs.

2.4 Delays, Loops, and Deadlocks

In general, looped directed-graph topologies are not allowed in SDF because they can cause data-dependency problems. However, if delay actors are inserted into loops, these problems can be fixed. Delay actors are special SDF actors that act as buffers for data-delay. They also act as actors for holding initial tokens during the start of the model execution.

It is convenient at this point to digress and talk about notation first. We use the notation Z^{-1} to denote delay actors in this paper. For those who are familiar with the conventions used in signal processing block diagrams, this will look familiar as the inverse of the Z transform variable. The value of exponent over the Z variable signifies the number of data delays in the actor. For example, Z^{-2} means an actor with two data delays and Z^{-3} means an actor with three data delays.

Let us examine an example of an SDF graph with delay actors. Consider the SDF graph shown in figure 10. Let us say that in the first iteration of execution, actor A produces tokens t_1 & t_2 on its first firing, t_3 & t_4 on its second firing, and t_5 & t_6 on its third firing. Now, since there is a delay actor in between actors A and B, it holds an initial token, which we will call t_0. From actor B's point of view, the tokens arriving during the iteration will be t_0, t_1, t_2 on the first firing and t_3, t_4, t_5 on the second firing. The token t_6 will not arrive until the next iteration. This illustrates a delay in the consumption of tokens by actor B from those tokens produced by actor A. The delay was caused by the insertion of an initial token, t_0, by the delay actor.

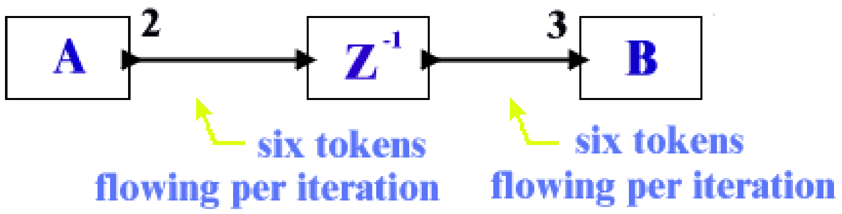

A
2
Z-1
3
B
six tokens flowing per iteration
six tokens flowing per iteration

Figure 10. An SDF graph with a delay actor

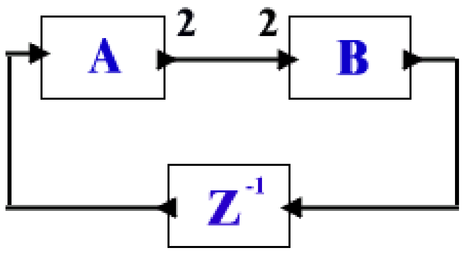


Figure 11. A looped SDF graph

Aside from having solutions to their balance equations, all schedulable SDF graphs should also be *deadlock*-free. Deadlock occurs when there are not enough tokens for any actor to execute at a certain point of time during the execution. For example, consider the SDF graph shown below, the delay actor produces two initial tokens during the start of the execution. Actor A then fires to produce three tokens for actor B. However, at this point actor B does not have enough tokens to fire. In fact, none of the actors in the SDF graph is ready to fire. This is deadlock. Usually, it is possible to fix deadlock problems by increasing the initial tokens in delay actors in the loop. For example, in figure 12, changing the delay actor from Z^{-2} to Z^{-4} is enough to prevent deadlock.

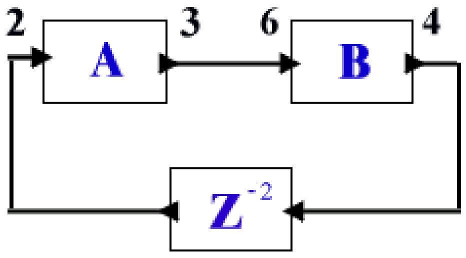


Figure 12. A deadlocked SDF graph

2.5 Design Features and DT Semantics

We came up with three important criteria in the design of DT. These criteria, in our opinion, form the core desirable properties of our proposed model of computation. These criteria also helped as our semantic guidelines in extending SDF.

a) *Uniform Token Flow*: The time interval between tokens should be regular and unchanging. This conforms to the idea of having sampled systems with fixed rates. The tokens flowing in DT do not keep internal time stamps. Each actor has to query the director in order to get the current simulation time.

b) *Causality*: Tokens produced by an actor should only depend on tokens produced or consumed in the past. This makes sense because we don't expect an actor to produce a token before it can calculate the token's value. For example, if an actor needs three tokens t_1, t_2, and t_3 to compute token t_4, then the time when tokens t_1, t_2, and t_3 are consumed should be earlier than or equal to the time when token t_4 is produced. Note that in DT semantics, time does not get incremented due to computation.

c) *SDF-style semantics*: We want DT to be a timed-superset of SDF with compatible token flow and scheduling. However, we will show in section 2.8 that we can only approximate this behavior. We will show that it is not possible to have uniform token flow, causality, and SDF-style semantics at the same time. Causality breaks for non-homogeneous actors in a feedback system. We have to introduce forced-latencies that preclude DT from being completely backward compatible with SDF. More on this later.

Side note: From here onwards, we will use the term 'DT dataflow graph' to interchangeably mean 'SDF graph'. Most of the topics we will discuss after this section have to do with temporal semantics, which are only interesting in DT dataflow graphs.

2.6 Global and Local Time

The firings of the actors in a DT dataflow graph form a partially ordered set. This execution ordering of the actors means that certain actors have to be fired before certain actors. However, since this ordering is only partial, some actors do not have execution ordering dependency with respect to other actors. This means that some of the actors in a DT dataflow graph can be executed concurrently. For example, if you consider the DT dataflow graph shown on figure 13, you will see that actor C does not have any data dependency relationships with actors A and B; hence actor C can be executed concurrently with actors A and B.

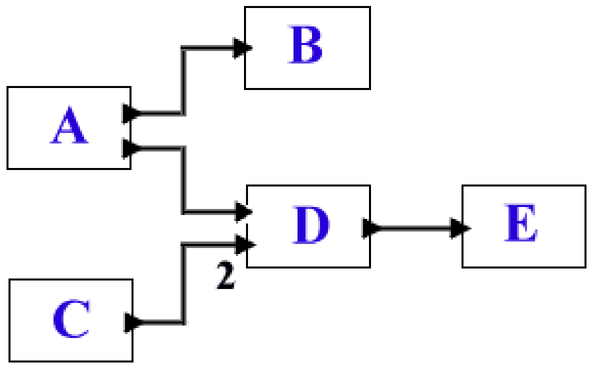


Figure 13. A DT dataflow graph illustrating concurrency and partial ordering

This inherent concurrency in DT dataflow graphs introduces two notions of time in DT – *global time* and *local time*. Global time increases steadily as execution progresses. Moreover, global time increments by fixed discrete chunks of time based on the value of the *period* parameter. On the other hand, local time applies to each of the ports in the model. All of the ports have distinct local times as an iteration proceeds. The local time of a port gets monotonically incremented every time a token is consumed. Local time obeys the following constraint with respect to global time:

$$\text{global time} \leq \text{local time} \leq (\text{global time} + \text{period})$$

The *period* parameter specifies how much simulated time one iteration takes to execute. For example, let us say that the DT dataflow graph shown in figure 13 has a period value of 3.5. At the start of the execution, the model has global time 0.0. At the end of the first iteration, the global time is 3.5. At the end of the second iteration, the global time is 7.0. This repeats ad infinitum.

2.7 Token Timeline Charts

In order to analyze discrete-time models, we need to introduce token timeline charts. For those who have seen or used Gantt charts for task scheduling, token timeline charts might look similar. Token timeline charts are useful for visualizing the time when tokens occur. This, in turn, helps in determining causality in DT dataflow graphs. These charts consist of token timelines for each edge of the DT dataflow graph. All token timeline charts are schedule-independent; i.e. the token timeline chart of a DT dataflow graph is independent of its schedule.

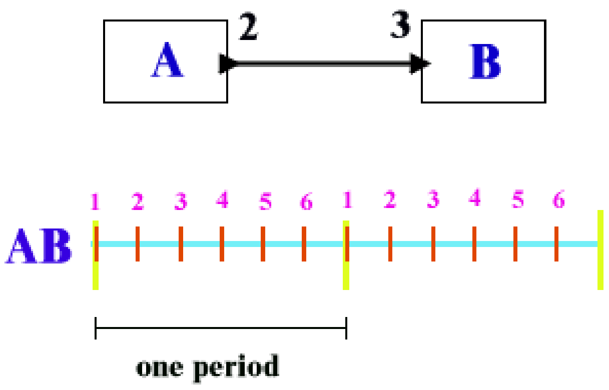


Figure 14. A simple DT dataflow graph (top) and its token timeline (bottom)

Consider the simple DT dataflow graph shown above. At each iteration, six tokens flow from actor A to actor B. These tokens are labeled 1 to 6 in the AB token timeline. There are two important things to note here. First, the time between tokens in the timeline is equally spaced. This is expected from design feature#1 in section 2.5. Second, the sum of these token intervals is equal to one period interval. This follows from DT semantics described in section 2.5. Using these observations, we can generalize and give the equation for the time interval between tokens in an edge:

time interval between tokens = period / token flow rate

Let us look at the slightly more complicated DT dataflow graph shown below. At each iteration, six tokens flow from actor A to actor B; and two tokens flow from actor B to actor C. If we view edges AB and BC as independent, we have the token timeline chart shown in figure 16. However, there is a data dependency between edge AB and edge BC. The token timeline chart shown in figure 16 is invalid because it doesn't take into account causality (design feature#2). Tokens cannot flow from actor B to actor C until after three tokens are made available to actor B by actor A.

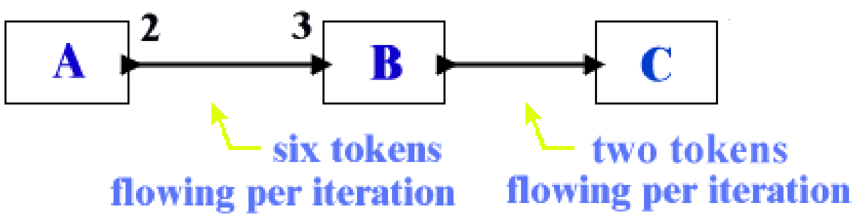

A
2
3
B
C
six tokens flowing per iteration
two tokens flowing per iteration

Figure 15. Another DT dataflow graph

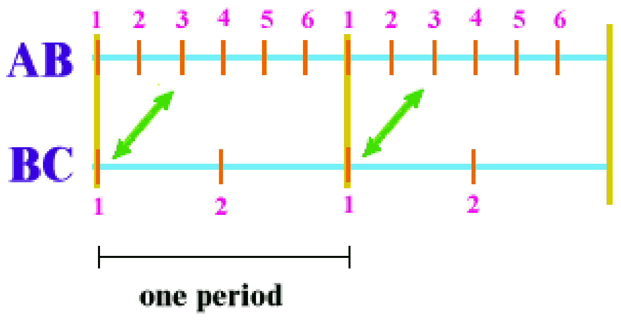


Figure 16. An invalid (non-causal) token timeline chart for the previous graph

Our first approach to keeping causality is through the introduction of initial time lags in the token timelines. We will show later in the next section that this approach has some flaws and will have to be abandoned. We are only showing it here to reveal the pitfalls we encountered while experimenting with DT; and to motivate the reader to the rationale behind our approach in next section. So in figure 17 we insert an initial time lag in the firing of actor B. This results in a corresponding initial time lag in the BC token timeline. This approach works for arbitrarily long chains of loop-less DT dataflow graphs. It also works for loop-less DT dataflow graphs with delay actors.

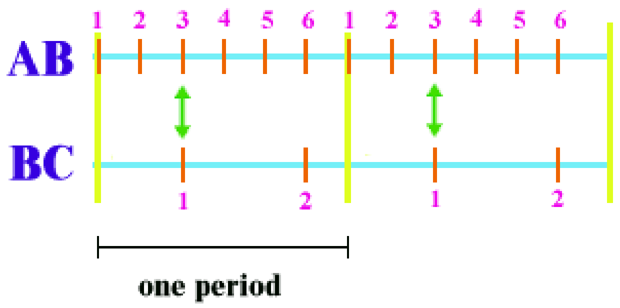


Figure 17. Token timeline chart corrected by inserting an initial time lag on edge BC

Figure 18 shows another DT dataflow graph and its token timeline chart. There are several interesting things to note regarding this DT dataflow graph. The graph has a delay actor. The result of the delay actor on the flow of tokens is apparent in the token timeline chart. We can see that there is an initial token inserted in the ZB timeline as a result of the delay actor. We can also see that the tokens in the ZB timeline are exactly those tokens in the AB timeline delayed by one token time interval. This observation results from one important DT fact: In DT, delay actors not only act as data-delay actors, but also as time-delay actors.

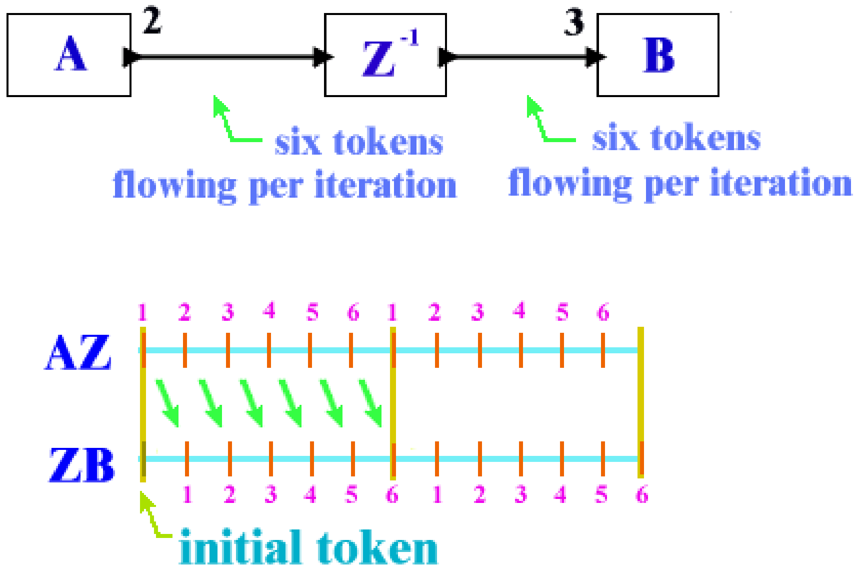


Figure 18. A DT dataflow graph with a delay actor

2.8 Causality and Latency

Inserting initial time lags is enough to fix all causality issues for loop-less DT dataflow graphs. However, problems invariably show up for looped DT dataflow graphs. In fact, these problems are serious enough that we have to abandon the idea of initial time lags and introduce the concept of latency.

Before we start talking about latency, let us first consider our original intuition regarding causality in looped DT dataflow graphs. We thought that being deadlock-free was enough for causality.

Conjecture: Every deadlock-free DT dataflow graph is causal

It turns out that our intuition is wrong. It is possible to have a deadlock-free DT dataflow graph that will have causality problems. Our counterexample to the conjecture is shown on figure 19. The graph is deadlock-free and has a unique schedule: ABZ. When we analyze the token flow across the actors through time, we will see that causality breaks. Let us begin our analysis. At first, the delay actor produces an initial token to cause actor A to fire. After which, actor A produces two tokens for actor B. We insert an initial time lag for actor B while it waits for the two tokens from actor A. When actor B fires, the output is fed to the delay token which causes a time delay before the token is fed back to actor A. This delay is actually longer than the time interval between tokens in actor A. Hence, actor A cannot fire in the second iteration without breaking causality (design feature#2) or breaking uniform token time intervals (design feature #1). This presents a serious problem in DT semantics. We've tried different approaches to solving this problem. One interesting approach involved abandoning uniform time intervals between tokens and introducing iteration start and end times. That ultimately didn't work because we kept on coming up with more complicated dataflow graphs like the one shown figure 20 that cause other problems.

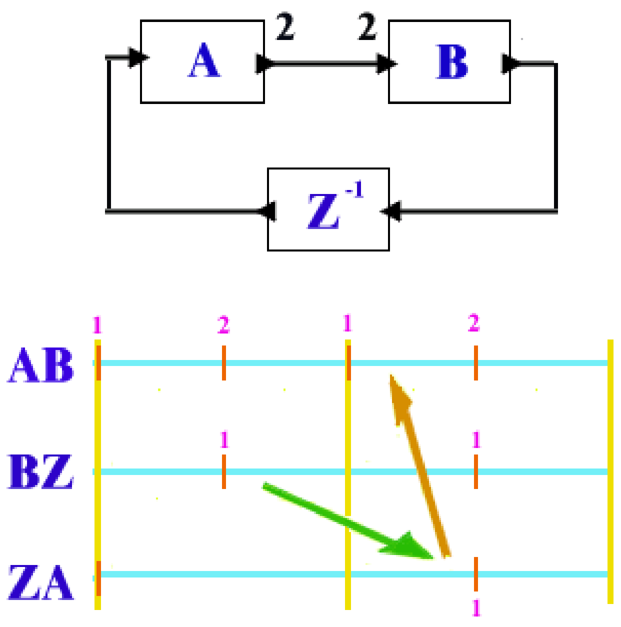


Figure 19. A counter-example to the conjecture. This non-causal DT dataflow graph cannot be made causal by inserting initial time lags

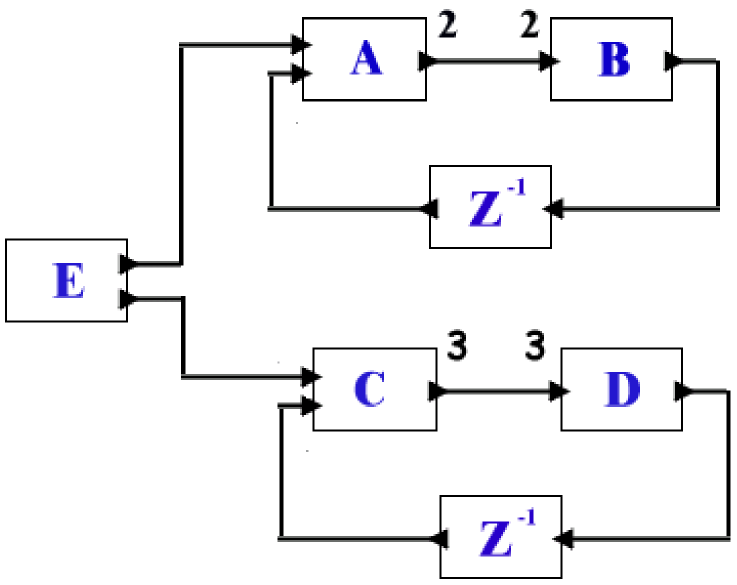


Figure 20. A more complicated DT dataflow graph

The ultimate solution we came up with is the introduction of *latency*. Latency comes in the form of initial tokens. Since we already observed that every actor with a non-homogeneous input port requires an implicit initial time lag, it might make sense to make the time lag explicit. Forcing initial tokens does this. We call these initial tokens as *latency tokens.*

Consider the example in figure 21. Instead of moving the BZ timeline to after the second token from actor A is available, we put an initial token in the output port of actor A, much like we have initial tokens on delay actors. The token timeline chart shows that this does not violate causality. Actor A can fire during the second iteration because of the latency token provided by actor B at the start of the first iteration. In general, the insertion of initial tokens on the output ports of actors with non-homogeneous inputs will get rid of all causality problems, even for looped topologies.

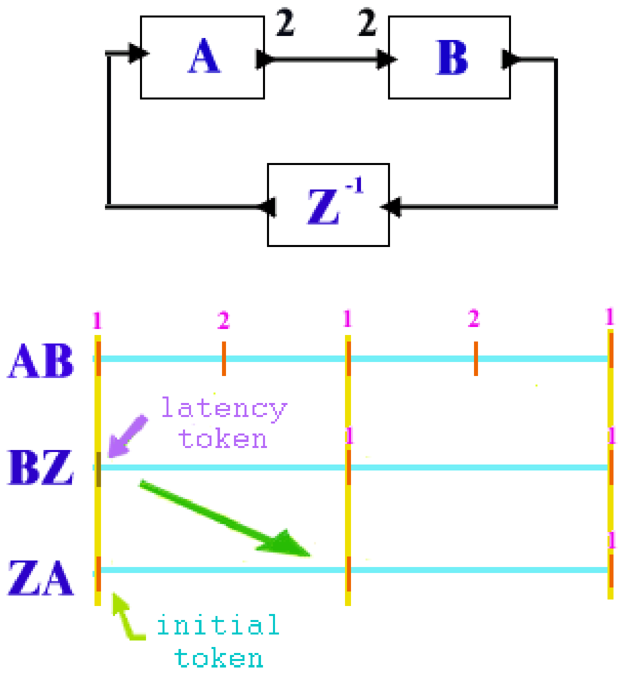


Figure 21. A fix for our counterexample using latency tokens

The introduction of mandatory latency precludes our backward-compatibility with SDF (design feature#3). This is not necessarily all that bad. In SDF, initial tokens are optional for non-delay actors. In DT, initial tokens are required for actors with non-homogeneous input ports. A properly designed SDF model will behave similarly under DT. The only difference would be at the start of the execution. For example, in audio signal processing applications, the difference between the execution of an SDF and a DT model might be a few milliseconds of silence.

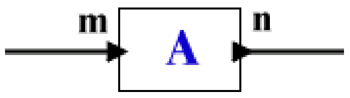


Figure 22. An SDF actor

How many initial tokens should be inserted on the output ports of non-homogeneous actors? Consider the actor shown in figure 22. Actor A has input port consumption rate **m** and output port production rate **n**. We have calculated that the number of initial tokens required for the output port should at least be:

$$\left\lceil \frac{n(m-1)}{m} \right\rceil$$

For SDF actors with multiple input ports, the same formula holds, but we replace the variable m with the maximum (argmax) of the input ports consumption rates. Please see the appendix for details on how we derived this expression.

Before we conclude this section, let us go back to the some of our previous DT diagrams and correct them by putting latency tokens on ports where these tokens are necessary. It turns out that we don't need to modify the token timeline charts for figures 14 and 18. However, we need to correct the token timeline chart of figure 15.

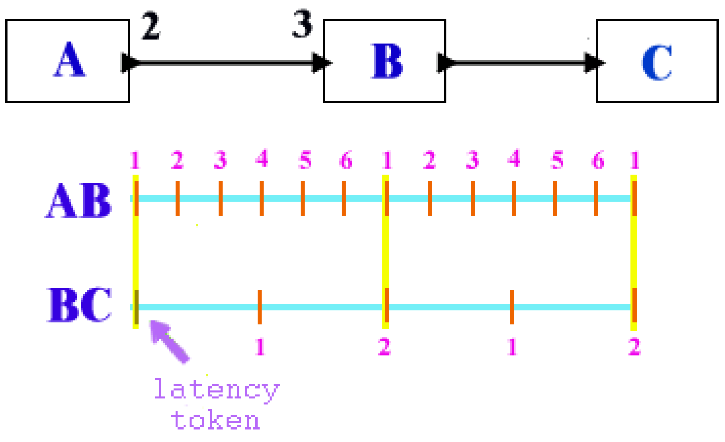


Figure 23. A corrected timeline for the graph in figure 15.

2.9 Hierarchical Discrete Time

Hierarchical DT has similar semantics as hierarchical SDF. The main difference has to do with temporal semantics. Before getting into this, it is first appropriate to discuss the details of hierarchical SDF. There are two ways to hierarchically compose SDF graphs. The composition can either be transparent or opaque.

Transparent composition of SDF graphs is trivial. The director unravels all transparent composite actors before execution and schedules the whole system as if it were not hierarchical. For example, consider the hierarchical SDF graph shown below. This graph is equivalent to non-hierarchical SDF graph shown below it. The same concept applies to the way DT handles transparent hierarchies.

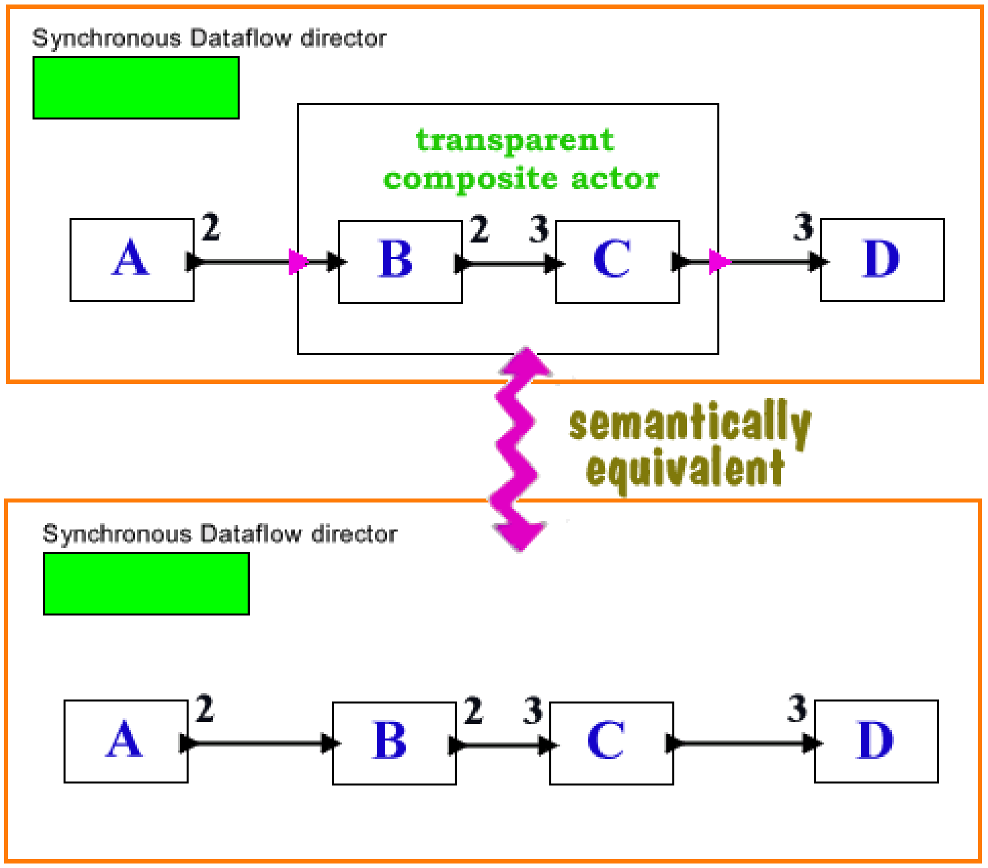

Synchronous Dataflow director
transparent composite actor
A
2
B
2
3
C
3
D
semantically equivalent
Synchronous Dataflow director
A
2
B
2
3
C
3
D

Figure 24. Two semantically equivalent SDF graphs. The top graph shows a hierarchical SDF graph with a transparent composite actor. The bottom graph shows a semantically equivalent non-hierarchical SDF graph.

Opaque composition of SDF graphs is more involved. The trick is coming up with schedules for both the inside and outside graphs that are consistent with one another. We first consider the scheduling of the graph inside the composite actor. The input and output ports of the composite actor are treated as actors with repetitions value of one and with unknown token production and consumption rates. The scheduler then uses balance equations to get the rates of these ports. Once the rates are determined, a schedule for the inside graph can be calculated. After which, a schedule for the outside graph can be calculated by treating the opaque composite actor as an ordinary atomic SDF actor with the recently computed rates on its input and output ports.

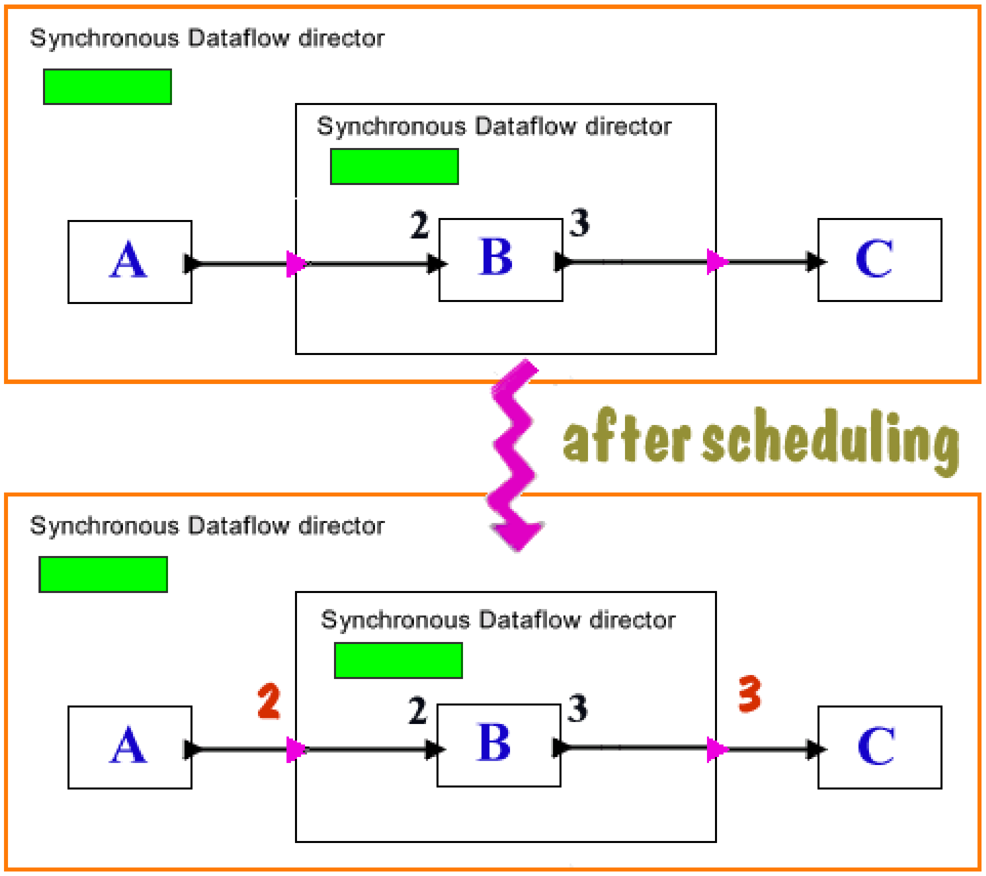

Synchronous Dataflow director
Synchronous Dataflow director
A
2
B
3
C
after scheduling
Synchronous Dataflow director
Synchronous Dataflow director
A
2
2
B
3
3
C

Figure 25. An example of an opaque hierarchical SDF graph

Consider the opaque hierarchical SDF graph shown above. After scheduling the inside graph, the port rates for the opaque composite actor containing actor B can be determined. In this example, the calculated token consumption rate is 2 and the calculated token production rate is 3. This information is then propagated to the outside graph. After which, it is possible to schedule the outside graph. The end result would be the schedule for the inside graph: B (a trivial schedule); and the schedule for the outside graph: (2A)X(3C). X stands for the composite actor containing actor B. It is important to note that the port rates are not just copied from the port rates of actor B. The scheduler actually had to solve balance equations to come up with token consumption rate and production rate values for the composite actor. The opaque hierarchical SDF graph in figure 26 shows another example of the procedure. The end result would be a schedule for the inside graph: (3B)(2C), and a schedule for the outside graph: (3A)(2X)(3D).

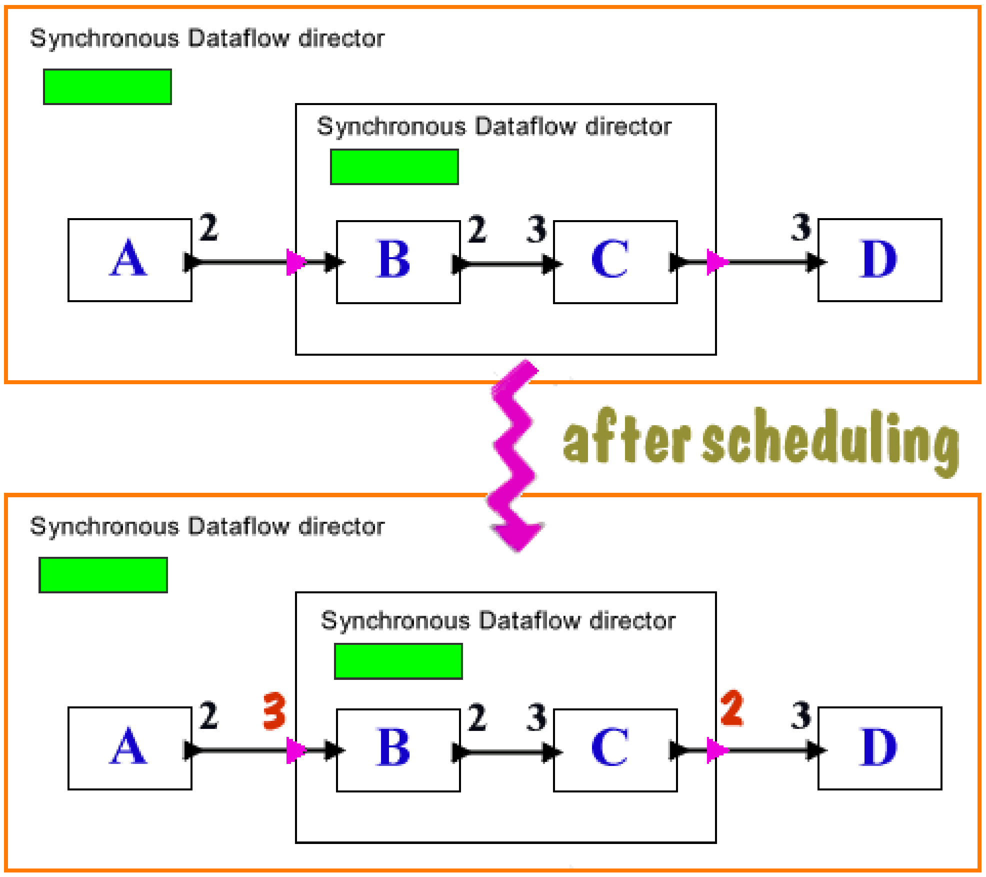

Synchronous Dataflow director
Synchronous Dataflow director
A
2
B
2
3
C
3
D
after scheduling
Synchronous Dataflow director
Synchronous Dataflow director
A
2
3
B
2
3
C
2
3
D

Figure 26. Another example of an opaque hierarchical SDF graph

Now that we have discussed the semantics of hierarchical SDF, it is time to discuss the temporal semantics of hierarchical DT. Actually, it is quite simple. The outer director acts a master and the inner director acts as a slave. Users cannot set the period parameter of the inner director's period parameter. The inner director uses the period of the outside director to determine its own period. This is why it is a slave to the outside director. There is an explicit formula for calculating the period of an inside director. Let us denote R_I as the actor repetitions of the composite actor that contains the inner director. The period of the inner director is given by the formula:

$$\text{inner period} = \text{outer period} / R_I$$

This formula was derived from a simple analysis of how to keep the token time intervals between the outside director and the inside director consistent.

3. DT and Other Ptolemy II Domains

3.1 Heterogeneous Hierarchies in Ptolemy II

In this section, we explain the semantics of composing DT with other Ptolemy II domains. In particular, we will discuss composing DT with SDF, DE, CT, and GR. We will provide simple examples of heterogeneous models involving DT. More elaborate examples and applications will be given in Section 4 of this paper. Note that, hierarchical compositions of DT with the CSP, PN, and DDE domains are not covered in this paper; and are the subject of future research.

3.2 Synchronous Dataflow (SDF) Domain

Hierarchically compositions of DT with SDF are very similar to plain hierarchical DT and plain hierarchical SDF. We already discussed these in section 2.9. There isn't much to mention except for the fact that the mixed model has temporal semantics that follows the DT model.

3.3 Discrete Event (DE) Domain

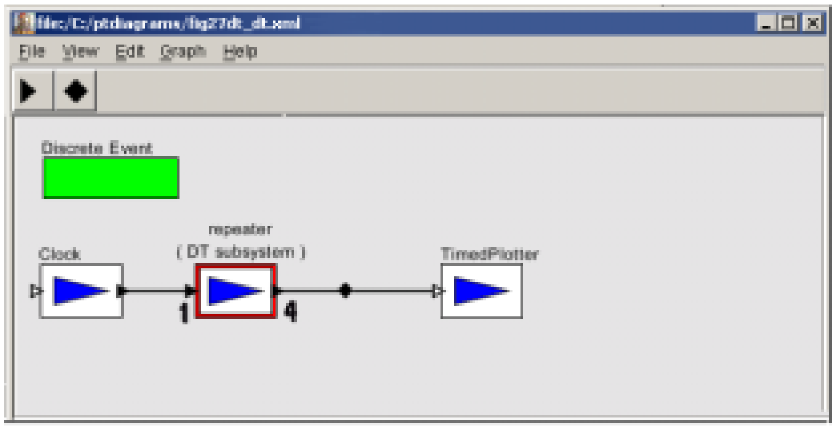
The Discrete Event (DE) domain [14] in Ptolemy II provides a general environment for time-oriented simulations of systems such as queuing systems, communication networks, and hardware systems. Discrete event systems are well understood by the simulation research community. There are several good books written on the subject, so we will not give a detailed account on the semantics of discrete event systems. Our main interest here is to explain the semantics of composing DT with DE.

Case#1: DT inside DE

The DT composite actor behaves like a clock actor. It requests the DE director to fire it periodically. The period parameter of the DT director is used to determine the time between firings. If the DT composite actor has input ports and there are not enough tokens on its input ports when it is fired, then it simply does not fire. If the DT composite actor has output ports, the times when tokens are produced are exactly those times when they show up on the token timeline chart of the DT dataflow graph. This means that the time between output tokens of each output port of the DT composite actor should be constant during a specific firing. This is the exact same behavior that we prescribed in design feature#1 of section 2.5. Figure 27 shows an example of a DT subsystem inside a DE model.

Case#2: DE inside DT

The DT director executes the DE composite actor as if it were a homogeneous atomic actor. The static DT schedule determines when the DE composite actor will be fired, so the DE composite actor cannot explicitly request to be fired at a certain time. To be more precise, the DT director ignores firing requests of a DE composite actor. The DT director acts as a master that controls the progression of time. The DE composite actor adjusts its time with respect to outside DT director.



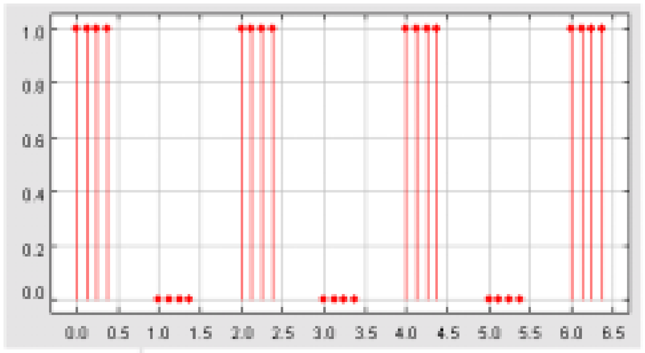
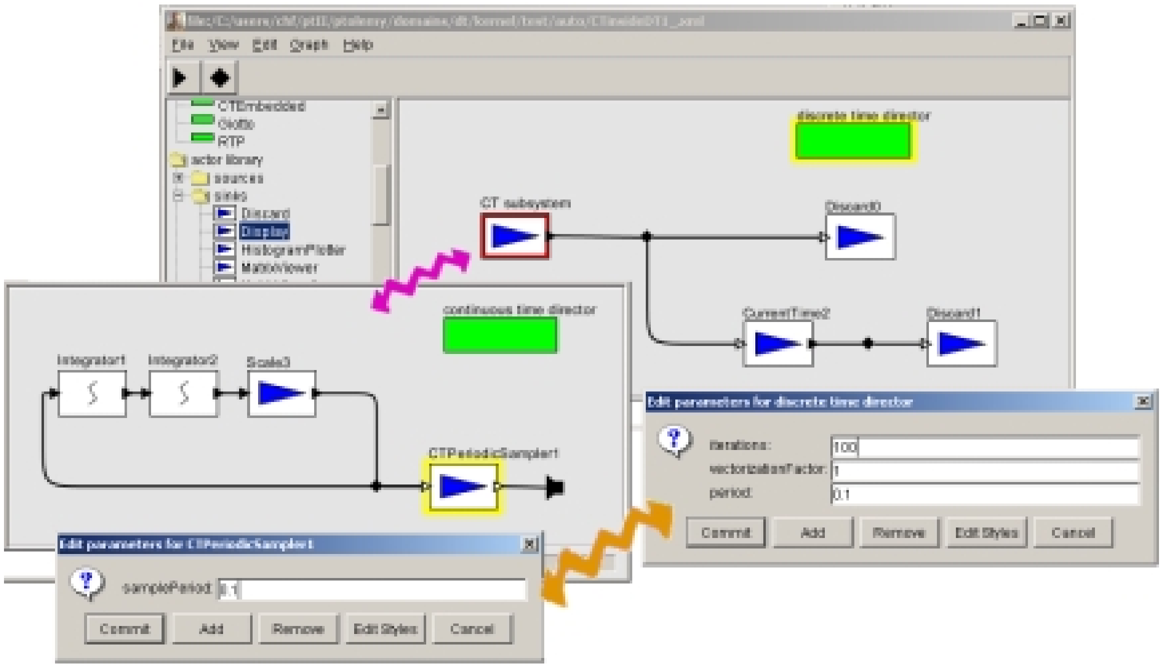


Figure 27. DT inside DE. The DT subsystem takes in values in its input port and produces four copies in its output port. The clock actor has period 2.0; and the DT subsystem has period 0.5 .



iterations: 100
vectorizationFactor: 1
period: 0.1

samplePeriod: 0.1

Figure 28. CT inside DT

3.4 Continuous Time (CT) Domain

The Continuous-Time (CT) domain [11] in Ptolemy II provides a general environment for representing and simulating systems governed by continuous-time behavior. It is frequently used in control system and mixed signal applications. Again, we refer the enthusiastic reader to the bibliography for more details.

Case#1: DT inside CT

The DT composite actor behaves like a clock actor. It requests the CT director to fire it periodically. The period parameter of the DT director is used to determine the time between firings. If the DT composite actor has input ports and there are not enough tokens on its input ports when it is fired, then it simply does not fire. If the DT composite actor has output ports, the times when tokens are produced are exactly those times when they show up on the token timeline chart of the DT dataflow graph. Note that, unlike the DE director, the CT director needs to fire its actors at times even when the actor didn't explicitly request for that firing. In view of this, if the DT composite actor is fired at a time that is not in its period intervals, it simply doesn't fire.

Case#2: CT inside DT

The semantics of running a CT composite actor inside a DT model is not well understood at this time. We are still doing research on this area. We do have a limited implementation, which we would describe here. The main problem we encountered has to do with static scheduling in DT. The CT composite actor needs to dynamically request its outside director to fire it at certain times. The DT director cannot handle dynamic firing requests. We need the dynamic dataflow (DDF) or process networks (PN) domains to handle such unpredictability.

The DT director executes the CT composite actor as if it were a homogeneous atomic actor. The static DT schedule determines when the CT composite actor will be fired, so the CT composite actor cannot explicitly request to be fired at a certain time. The DT director acts as a master that controls the progression of time. The CT composite actor adjusts its time with respect to outside DT director. These restrictions dictate a limitation on the CT subsystem. The general CT subsystem will not work in DT. In particular, we need to impose the restriction that the CTPeriodicSampler actor inside the CT subsystem has to have a sample period that matches the period of the DT director. This is shown in figure 28.

3.5 Finite-State-Machine (FSM) Domain

The Finite-State-Machine (FSM) domain in Ptolemy II provides a general environment for representing control-flow graphs. It is frequently used in control-oriented applications. Finite-state-machines are well studied and understood by the general computer science community. In fact, it is the most basic model of computation covered in an undergrad complexity theory class.

Case#1: FSM inside DT

Since FSM is an untimed domain, putting an FSM composite actor inside DT is simple. The static DT schedule determines when the FSM composite actor will be fired. Under DT, a FSM composite actor would behave like a homogeneous atomic actor. We can think of the FSM composite actor as an actor capable of control-flow as well as dataflow.

Case#2: DT inside FSM

This case is currently not implemented in Ptolemy II. This case is actually a special case of *CHARTS which will be covered in case#3. It is the subject of future work.

Case#3: *Charts

*Charts [5] (pronounced as "star-charts") form one of the most important hierarchical compositions in Ptolemy II. In general, *charts are arbitrarily deep hierarchical compositions of mixed concurrency models (such as dataflow, CSP, SR, DE) with FSM's placed anywhere within it. This general case is currently not implemented in Ptolemy II. There has been some work on heterochronous dataflow (HDF), which is SDF inside FSM inside SDF. There are plans to extend this to DT-HDF (DT inside FSM inside DT) in the future.

3.5 Graphics (GR) Domain

GR is a new experimental domain under development in Ptolemy II. Its primary purpose is to provide an infrastructure for displaying three-dimensional graphics in Ptolemy II. GR. is an untimed domain that follows loop-less synchronous/reactive (SR) semantics. It is useful to hierarchically combine DT with GR to produce animated simulations. Both domains work hand in hand to produce well-defined Ptolemy models for a visual simulation. GR handles the visual display of information; and DT handles the passage of time.

Case#1: DT inside GR

The DT composite actor behaves like a domain-polymorphic actor. It fires when the GR scheduler determines that it needs to be fired. The period parameter of the DT director is used to keep track of time. If the DT composite actor has input ports and there are not enough tokens on its input ports when it is fired, then it simply does not fire. If the DT composite actor has output ports, the times when tokens are produced are exactly those times when they show up on the token timeline chart of the DT dataflow graph.

Case#2: GR inside DT

Since GR is an untimed domain, hierarchical compositions with DT are simple. Under DT, a GR composite actor would behave like a homogeneous atomic actor. We can think of the GR composite actor as a glorified plotter that takes in data and produces graphical output.

4. Applications in 3D Simulation

4.1 GR Revisited

As mentioned previously, GR is a new experimental domain under development in Ptolemy II. We will give more background information on GR in this section because it plays a key role in the applications that we will discuss. The basic idea behind the GR domain is to arrange geometry and transform actors in a directed-acyclic-graph to represent the location and orientation of objects in a natural world scene. This topology of connected GR actors form what is commonly called in computer graphics literature as a *scene graph*. The GR director converts the GR scene graph into a Java3D [1] representation for rendering on the computer screen. It is convenient to talk about two types of scene graphs at this point. The first type is the abstract scene graph. The second type is the GR scene graph used in implementation. Abstract scene graphs provide simple and intuitive view of the overall structure of a scene, but do not provide complete information. GR scene graphs are verbose and hard to read, but contain all the information needed by the computer to render a scene. We will talk about these later. Meanwhile, let us discuss abstract scene graphs further. Figure 29 shows the abstract scene graph of a robotic arm and a tower of Hanoi set on top of a table. The abstract scene graph captures the parent-child relationship between different objects in the scene. For example, when the upper limb of the robotic arm moves, the end-effector of the robotic arm moves with it. In other words, the upper limb is a parent of the end-effector.

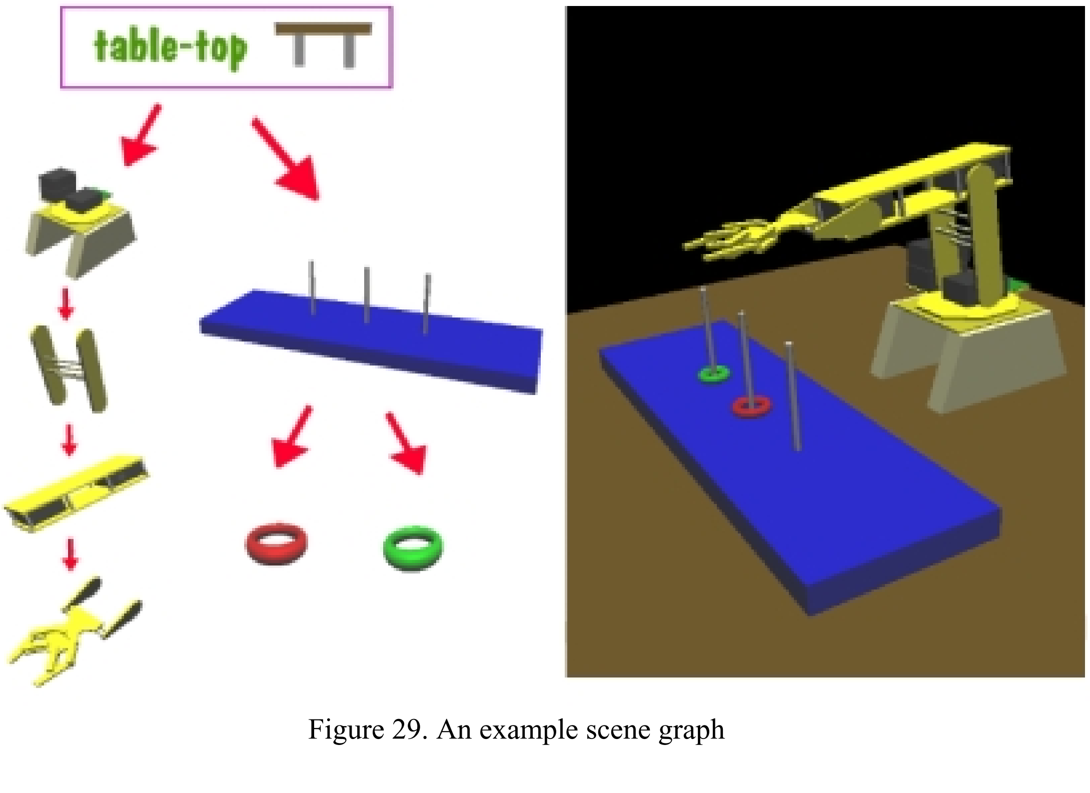
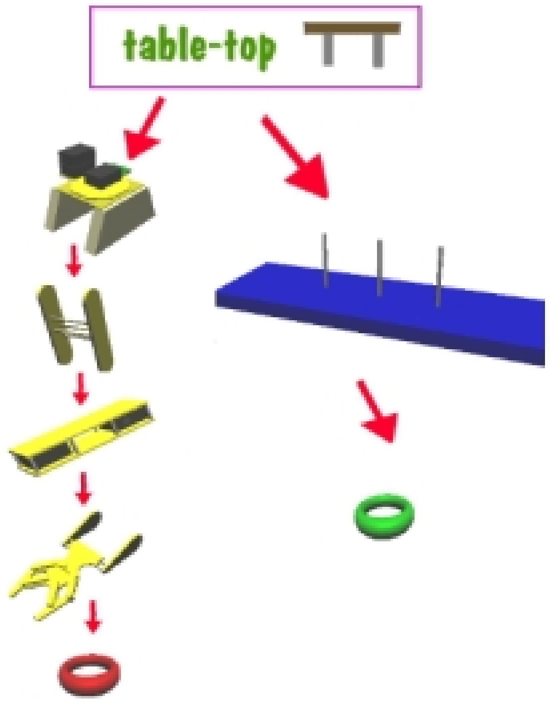


Figure 29. An example scene graph

One key point about scene graphs is that the graph's topology may change over time. For example, the abstract scene graph in figure 29 may change into the one in figure 30 when the robotic arm grabs the red ring. In this case, the red ring is no longer a child of the tower of Hanoi set. In general, in a complex scene graph, there can be arbitrary pruning and grafting of subgraphs during a simulation.



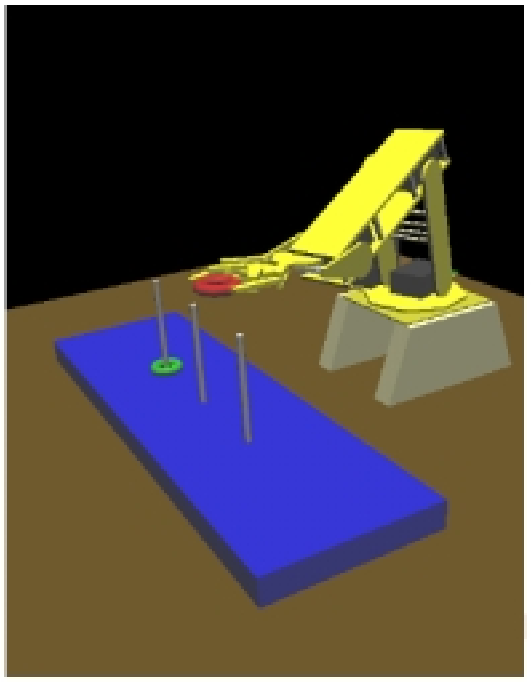


Figure 30. Scene graphs may change over time

4.2 Robotic arm model

Most of the 3D simulation applications that will be discussed in this paper will be related to a particular robotic arm. We decided to work with a Lynxmotion 5-axis robotic arm in the course of the development of the GR domain. In our opinion, robotic arms offer a nice mixture of electromechanical complexity that makes it an ideal application of our current research. Figure 31 shows a picture of the robotic arm on the left and its simulated counterpart on the right. The robotic arm has four degrees of freedom and an end-effector in the form of a gripper. We used Ptolemy II to control the movement of the robotic arm through serial communication with an embedded PIC16c620 microcontroller. The microcontroller handles the low level PWM control of the arm's servomotors. Ptolemy II handles high-level functions like motion planning and inverse kinematics. In addition, Ptolemy II uses the GR domain to simulate the visual appearance and movement of the robotic arm.

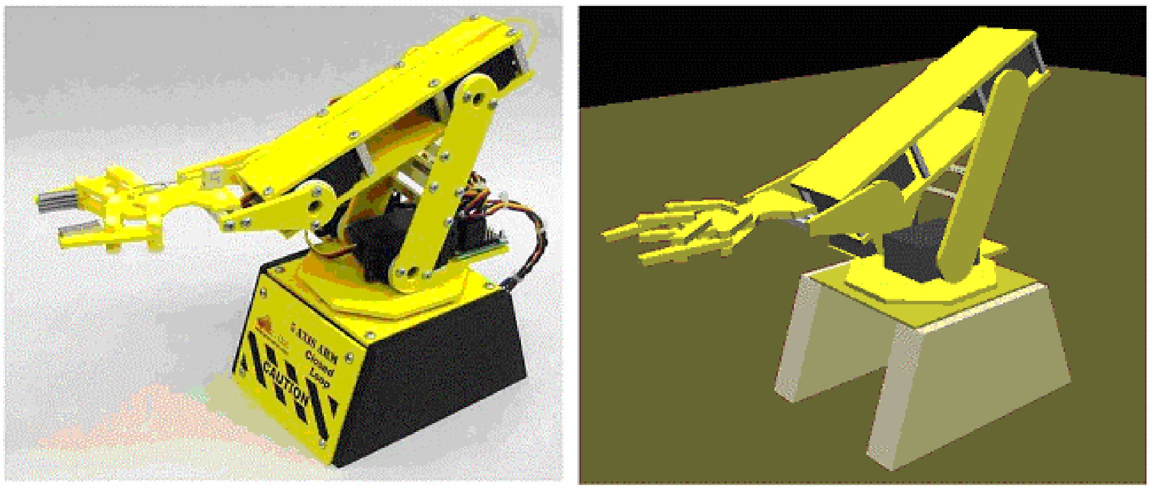


Figure 31. Real (left) and simulated (right) robotic arm

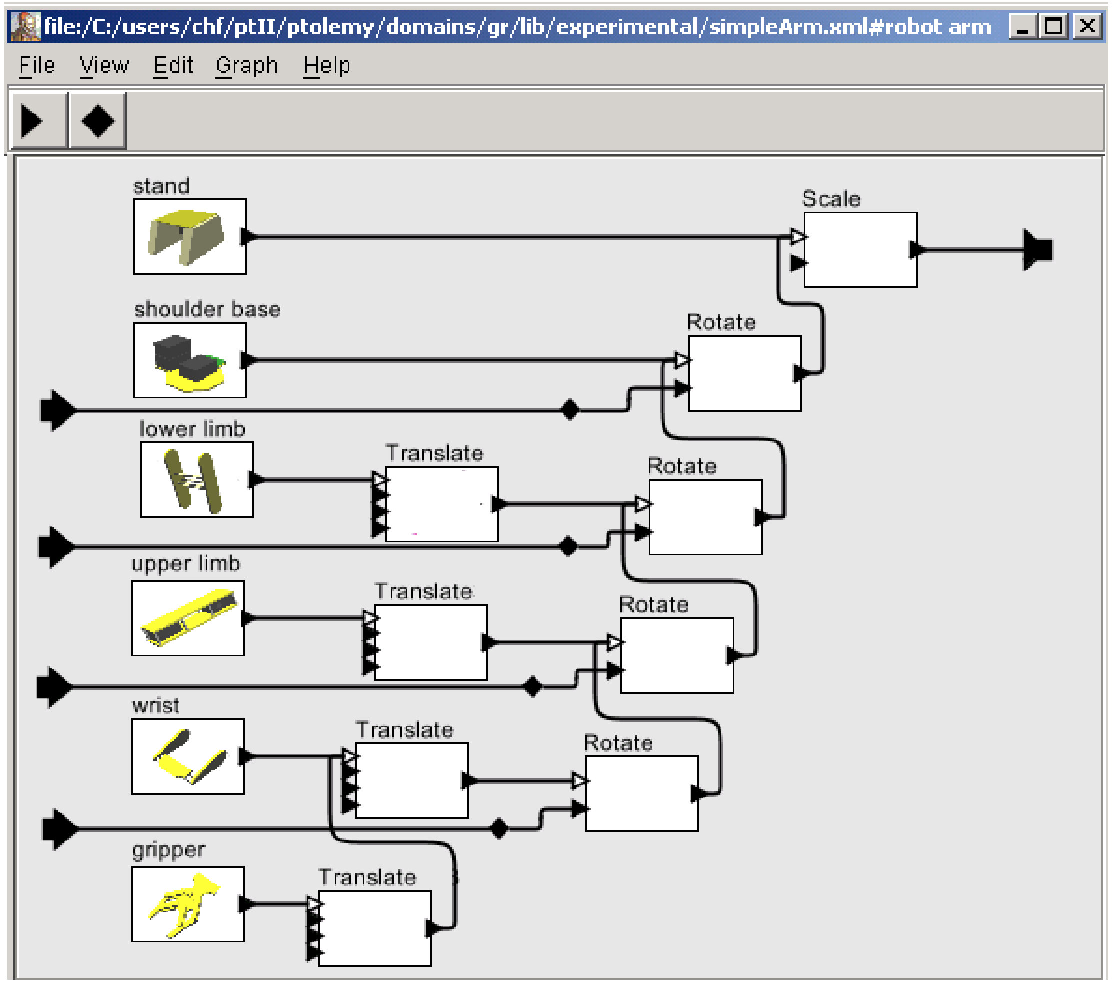


Figure 32. The GR scene graph for the robotic arm

It is now convenient to return to our discussion about GR scene graphs. Scene graphs are represented internally in the GR domain through geometry and transformation actors. The geometry actors contain polygon vertices that represent the mechanical shape of the object to be simulated. The transformation actors contain 4x4 matrices that are used to transform the polygon vertices of the geometry actors. There are currently three transformation actors in Ptolemy II – rotate, translate, and scale. The GR scene graph of our robotic arm model is shown in figure 32. There are six geometry actors in the model – robotic arm stand, shoulder base, lower limb, upper limb, wrist, and end-effector. These actors actually refine to simpler geometry actors, but that is not important in this discussion. Instead, it is important to note that there are transform actors that are used for the pose and placement of the geometry actors. This is what differentiates the GR scene graph from the abstract scene graph.

4.3 DT & CT: Pendulum Model

Up to now, DT has been largely missing in our applications discussions. It is time to show applications that use DT. The GR domain by itself is not enough for producing interesting simulations in Ptolemy II. It is necessary to bring other Ptolemy II domains to the forefront to drive the 3D simulations in GR. Figure 33 shows a top-level representation of a simple pendulum model. The discrete-time model refines to a CT subsystem and a GR subsystem. The CT subsystem models the physical characteristics of the pendulum. The GR subsystem models the visual appearance of the pendulum. The top-level DT system acts as the overall coordinator and timekeeper.

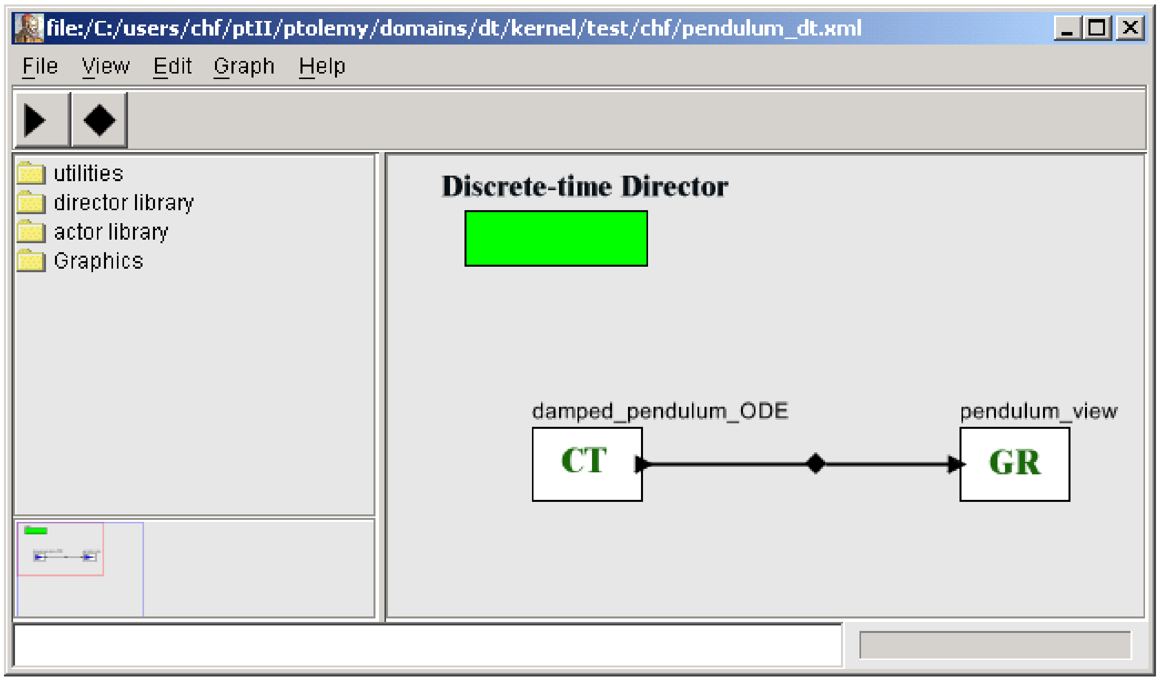
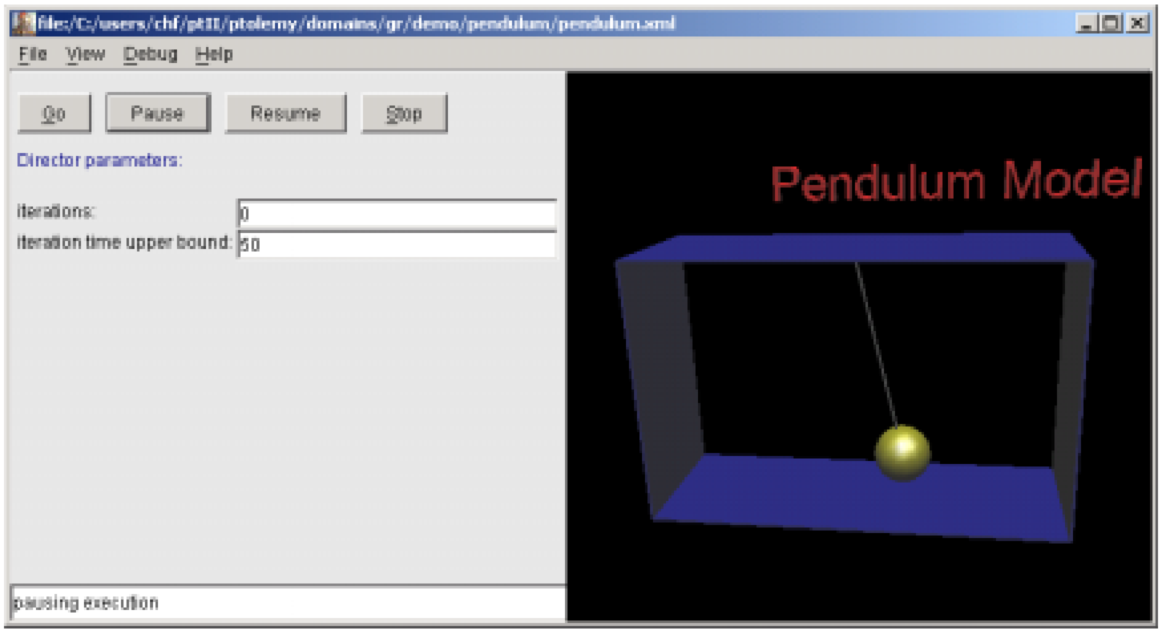

file:/C:/users/chf/ptII/ptolemy/domains/dt/kernel/test/chf/pendulum_dt.xml

Figure 33. Top-level representation of the pendulum model


file:/C:/users/chf/ptII/ptolemy/domains/gr/demo/pendulum/pendulum.xml
File View Debug Help
Go
Pause
Resume
Stop
Pendulum Model

Director parameters:

iterations: 0

iteration time upper bound: 50

pausing execution

Figure 34. Vergil screenshot of the pendulum animation

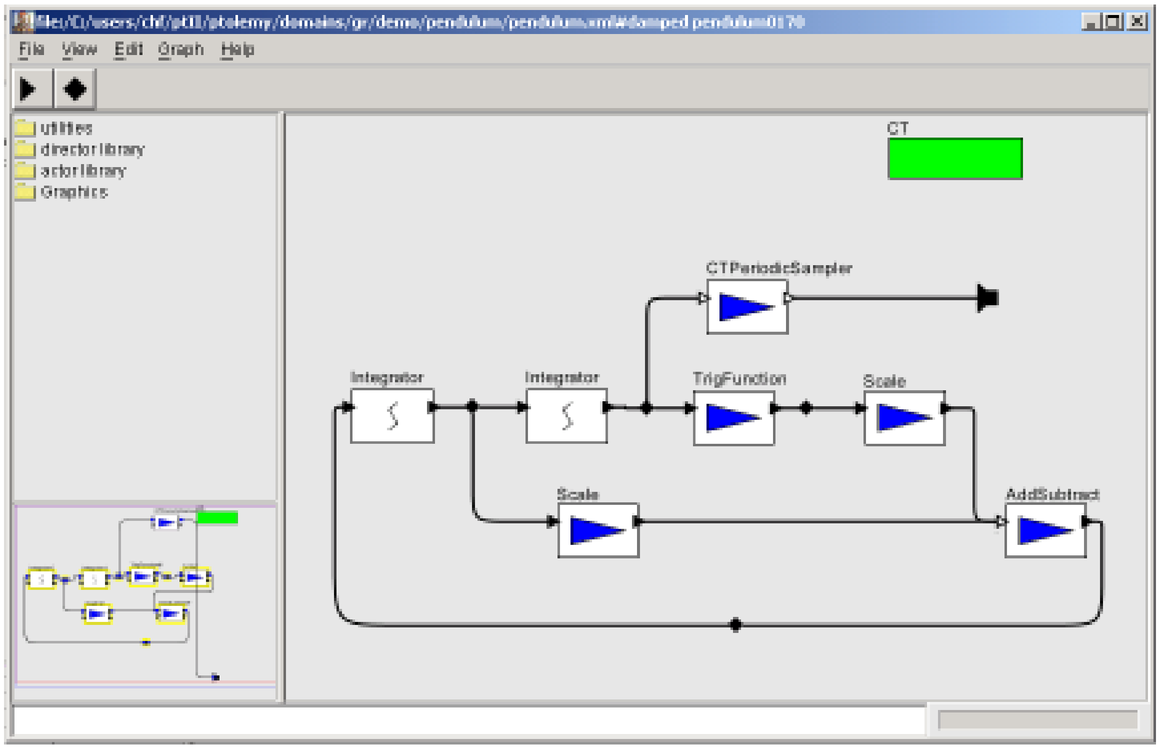


Figure 35. Continuous-time model for the pendulum

The CT subsystem drives the angular displacement of the pendulum as time proceeds. Figure 35 shows the CT subsystem that characterizes damped pendulum dynamics. For those of you familiar with Simulink, this diagram will look like with Simulink .mdl model. This diagram is actually equivalent to the second order differential equation:

$$\frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \frac{g}{l}\sin\theta = 0$$

where θ is the angular displacement of the pendulum; t is the time; q is the damping factor; g is earth's gravitational acceleration; and l is the length of the pendulum string.

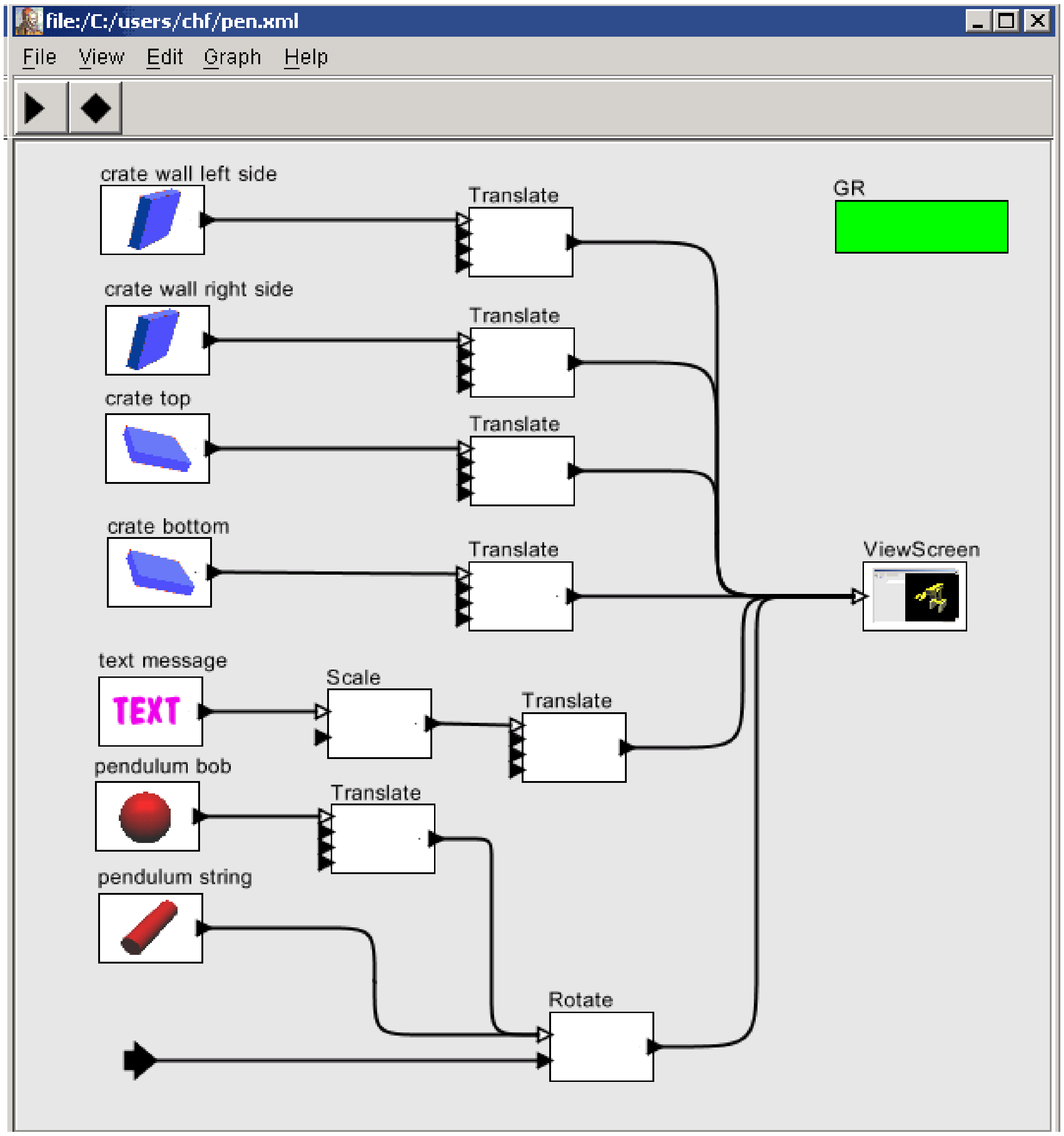


Figure 36. The GR scene graph for pendulum model

The GR scene graph for this pendulum model is very simple. It is shown in figure 36. Box-shaped geometry actors represent the four walls of the crate. The cylinder geometry actor represents the pendulum string. The sphere geometry actor represents the pendulum bob. It is important to note that the pendulum bob is a child of the pendulum string. The rotate transform actor gets angular displacement values from the input port and determines how the pendulum should be rendered. The Viewscreen actor is used as the root of the GR scene graph.

4.4 DT & DE: Inverse Kinematics

One of the most important aspects of controlling a robotic arm has to do with the positioning of the end-effector of the robotic arm. In general, given a desired position for the end-effector, the programmer of a robotic system does not want to explicitly specify the joint torques for each motor of a robotic arm. Instead, the programmer wants an algorithm that would automatically calculate the joint torques. This problem is called the *inverse kinematics* problem. Inverse kinematics is a well-studied and well-understood area of robotics. There are numerous standard inverse kinematics algorithms in the robotics and computer graphics literature [6][7][17]. For our purposes, we just want to pick an off-the-shelf inverse kinematics algorithm and use it in our robotic arm simulations. Inverse kinematics is a vital prerequisite to building more complicated simulation models of robotic arms. For example, if we want to have an unscripted animation of the autonomous movement of a robotic arm, we need to specify a space curve and apply inverse kinematics at key points in that space curve.

We chose to implement the *cyclic-coordinate descent* inverse kinematics algorithm. It is a relatively simple iterative algorithm that can be implemented using dataflow models. However, it requires some control flow parts that cannot be modeled using DT. We opted to use the discrete event (DE) domain and hierarchically compose it with DT in our final implementation. DT handles the parts of the algorithm that can be modeled using static dataflow. These include calculation of dot products & cross products, saturated arithmetic, and vector rotations.

We will not discuss the algorithmic details of the cyclic-coordinate descent algorithm here. Instead, we would like to give screenshots of the implementation and results. Figure 37 shows a DE subsystem that further refines to several DT subsystems. Figure 38 shows a screenshot of a demo that uses our implementation of the cyclic-coordinate descent algorithm. One minor side note here – although the block diagram of the cyclic-coordinate descent algorithm implementation looks very complicated, it is actually very simple. We did not vectorize the connections in our model. We can easily make the model less cluttered by using the RecordAssembler and RecordDisassembler actors in Ptolemy II.

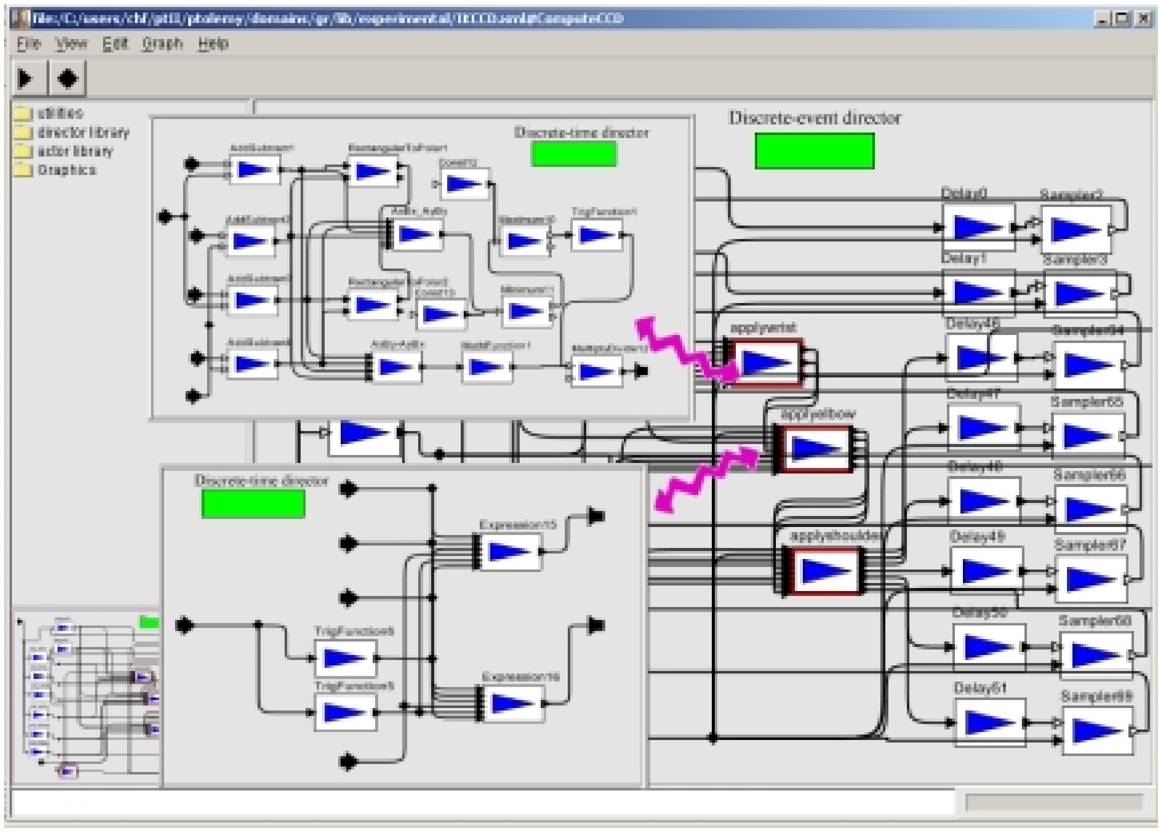
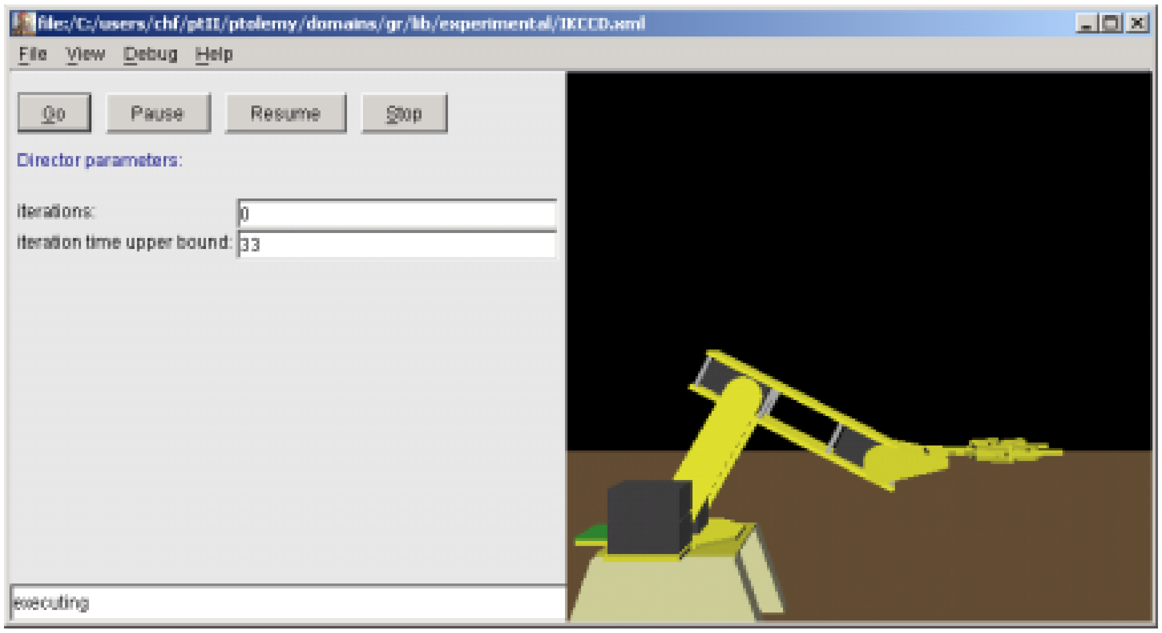


Figure 37. Ptolemy II screenshot of cyclic-coordinate-descent algorithm implementation using a hierarchical composition of DT and DE


file:/C:/users/chf/ptII/ptolemy/domains/gr/lib/experimental/IKCCD.xml
File View Debug Help
Go
Pause
Resume
Stop

Director parameters:

iterations: 0

iteration time upper bound: 33

executing

Figure 38. A screenshot of the inverse kinematics demo

4.5 DT & FSM: Picking Objects

We used the *charts formalism [5] to represent the way a robotic arm would pick and drop objects. Recall from figures 29 and 30 that the scene graph of a simulation may change over time. For a simple system like a tower of Hanoi set with two rings, there are three possible scene graphs. These scene graphs correspond to three states in the robotic arm simulation. The states are shown in figure 39. The *chart representation enabled us to animate the robotic arm picking and moving rings around in the tower of Hanoi set.

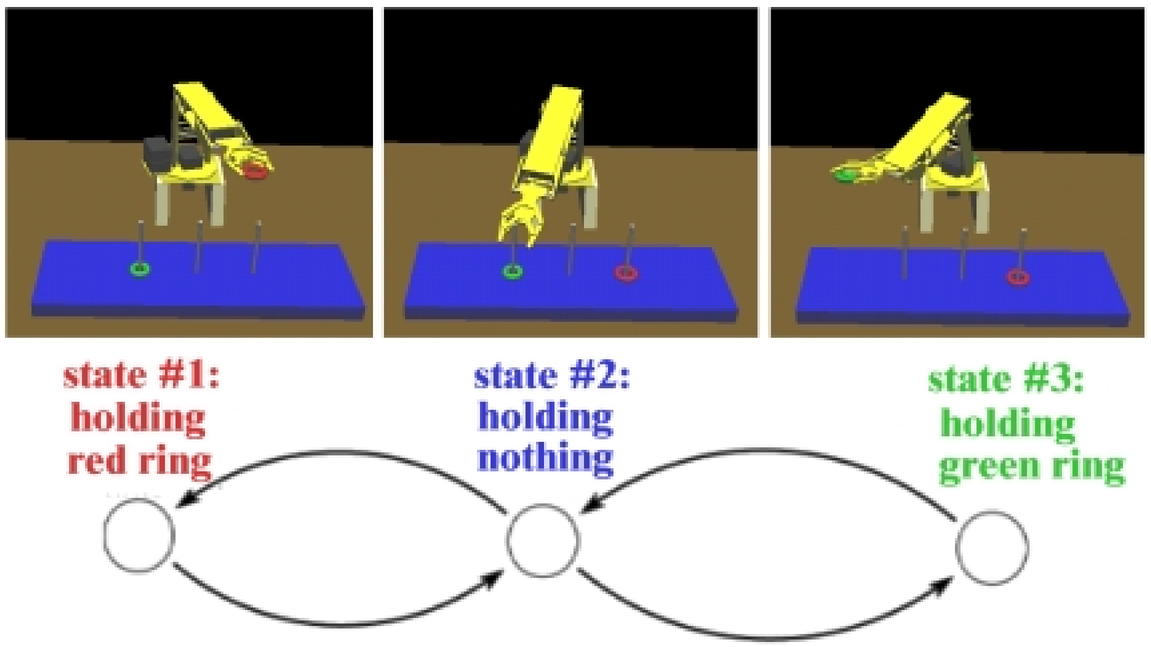

state #1: holding red ring
state #2: holding nothing
state #3: holding green ring

Figure 39. Three states for the robotic arm animation

Figure 40 shows the diagram for our robotic arm animation. In particular, it shows the hierarchical composition of DT with FSM and with GR. The FSM subsystem stores the internal state of the robotic arm. As mentioned previously, there are three states in this subsystem. The GR subsystem stores the scene graph for the robotic arm model and renders the animation on the computer screen. The top-level DT system controls the overall progression of time.

To sum up this section, we would like to reiterate that DT is useful in acting as a timekeeper in our 3D simulations. DT does not have the run-time overhead of DE and CT, so it is beneficial to use it in producing real-time animations.

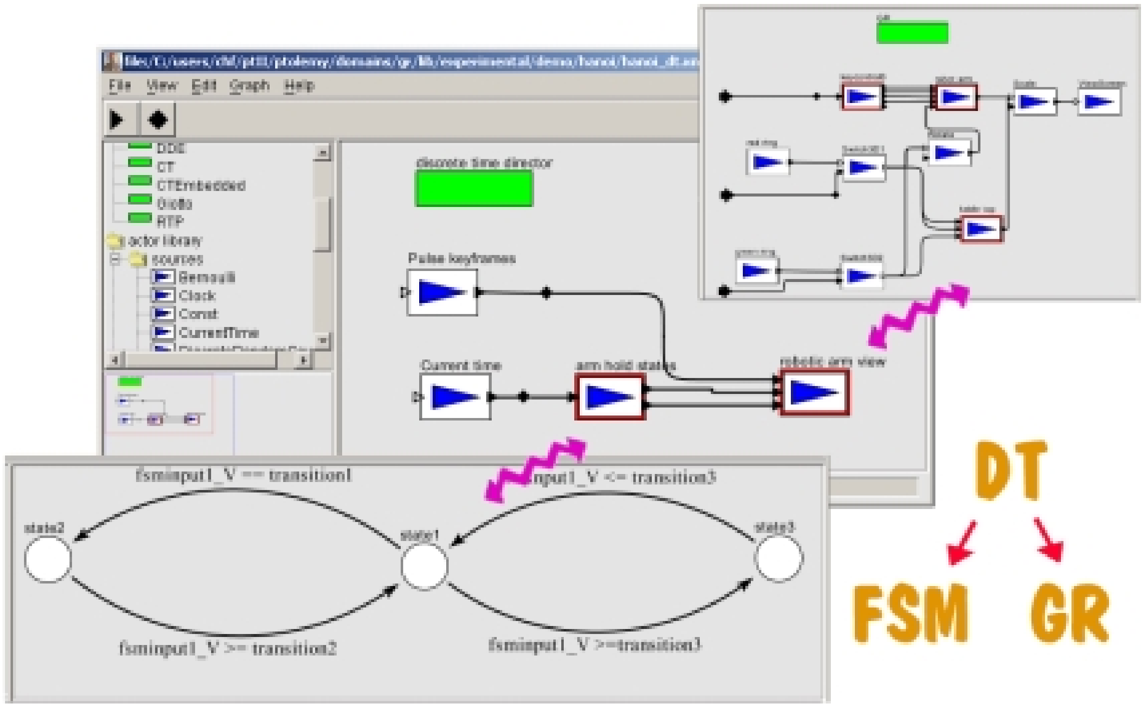


Figure 40. Nested hierarchies: DT refining to FSM and GR

5. Conclusion & Future Work

DT is a timed-extension of SDF with mandatory latencies for actors with non-homogeneous input ports. DT provides temporal semantics to SDF, which makes DT better suited for hierarchical composition with DE and CT. The main application of DT is the area of systems and signals. However, we have competently used it for 3D simulation applications. DT, when hierarchically combined with GR, is useful for providing temporal semantics to interactive computer animations.

There is certainly a lot of room for future work in this research area. We have barely scratched the surface in our study of discrete-time models and its applications. Here is a sampling of research directions that are worth pursuing in the future:

- The current implementation of the discrete-time domain in Ptolemy II is not optimized for speed. A lot of the code was written without thorough understanding of the powerful features and quirks of the Ptolemy kernel. It is quite possible that a rewrite of the major portions of DT is necessary to gain improvements in simulation speeds.
- There is a code-generation framework for the SDF domain in Ptolemy II. This framework compiles SDF models into Java programs. Extending this framework to work with DT shouldn't be very difficult.
- There are several domains in Ptolemy II that are left unexplored in the context of hierarchical compositions with DT. These domains include CSP, DDE, and PN. Also, there is still a lot of work to be done in the general case of hierarchically linking FSM with DT.
- On the applications side, we can implement inverse kinematics algorithms using Jacobian matrix methods. This method of doing inverse kinematics may be the better suited for the discrete-time domain.

6. References

[1] D. Bouvier, "Getting Started with the Java 3D API", Sun Microsystems. Mountain View, CA. 1999

[2] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems," Int. Journal of Computer Simulation, special issue on "Simulation Software Development", vol. 4, pp. 155-182, April, 1994.

[3] C. Cassandras, "Discrete Event Systems: Modeling and Performance Analysis", Aksen Associates Incorporated Publishers. Boston, MA. 1993

[4] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. "Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java". Memorandum UCB/ERL M99/44, EECS, University of California, Berkeley, July 19, 1999.

[5] 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.

[6] J. Lander, "Oh My God, I Inverted Kine! ", Game Developer magazine, pp. 9-14, September 1998.

[7] J. Lander, "Making Kine More Flexible", Game Developer magazine, pp. 15-22, November 1998.

[8] E.A. Lee and A. 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

[9] E.A. Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", IEEE Trans. On Computers, January 1987

[10] E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow", Proceedings of the IEEE, vol. 75, no. 9, pp. 1235-1245, September 1987.

[11] E.A. Lee and T.M. Parks, "Dataflow Process Networks", Proceedings of the IEEE, vol. 83, no. 5, pp. 773-801, May 1995

[12] J. Liu, "Continuous Time and Mixed-Signal Simulation in Ptolemy II", MS Report, UCB/ERL Memorandum M98/74, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998

[13] B. Mirtich, “Timewarp Rigid Body Simulation”, SIGGRAPH Computer Graphics Proceedings 2000, pp. 193-200, July 2000.

[14] L. Muliadi, “Discrete Event Modeling in Ptolemy II”, MS Report, Dept. of EECS, University of California, Berkeley, CA 94720. May 1999.

[15] P. Murthy, S. Bhattacharyya, and E.A. Lee. “Joint Minimization of Code and Data for Synchronous Dataflow Programs”, Journal of Formal Methods in System Design, Vol. 11, No. 1, July 1997.

[16] J. Pino. “Cosimulating Synchronous DSP Designs with Analog RF Circuits”. Ptolemy miniconference 1999, http://ptolemy.eecs.berkeley.edu/conferences/99/pino.pdf

[17] C. Welman, “Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation”. Master’s Thesis, Simon Fraser University. 1993.

Appendix: Non-homogeneous DT actors

In Section 2.8, we discussed a necessary criterion for all non-homogeneous actors. Specifically, we imposed that initial tokens should be inserted on the output ports of non-homogeneous actors. Consider the non-homogeneous actor shown in figure 41. It has an input port consumption rate of value *m* and an output port production rate of value *n*.

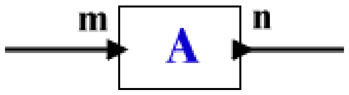


Figure 41. An SDF actor

The number of initial tokens needed for this actor should at least be

$$\left\lceil \frac{n(m-1)}{m} \right\rceil$$

where the half-brackets stand for the integer ceiling function. We will now discuss how we arrived at this value.

As a matter of convenience, we enlarge the topology in figure 41 into the one shown in figure 42. Figure 42 shows the same actor A with the same port rates. However, we've added some dummy actors C and B to act as the actors that feed and get fed by actor A. With this we can talk about arcs CA and AB. Note that it doesn't matter what the port rates of actors C and B are. It also doesn't matter whether actors C and B are connected to other actors as part of a larger graph (shown as wavy orange curves). We are really only interested in actor A and the tokens that flow across arcs CA and AB. Essentially, we want tokens flowing across arc AB to have a causal relationship with the tokens flowing across arc CA.

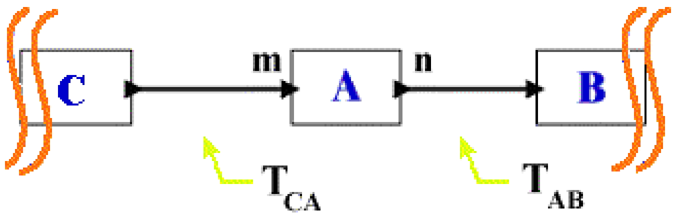


Figure 42: an SDF actor as part of an SDF graph

Before we continue, let us define some important variables that we need to use in our discussion. Let *P* denote the period parameter of the DT system. Let *R* denote the repetitions of actor A in the schedule. Now we can calculate the time interval between tokens in arc CA, which we shall call T_{CA}. Likewise, we can calculate the time interval between tokens in arc AB, which we shall call T_{AB}. These equations follow straight from sections 2.4 and 2.7

$$T_{CA} = \frac{P}{m\,R} \qquad\qquad T_{AB} = \frac{P}{n\,R}$$

In order to ensure causality, we have to put initial tokens (also called *latency*) on actor A. These initial tokens enable actor A to produce tokens at the start of an iteration, thereby eliminating the need for explicit initial time lags (discussed in section 2.7). However, we need to determine how many initial tokens are enough to ensure causality. Let us denote this number of initial tokens as *k*. Each initial token on the output port of actor A essentially causes an implicit initial time lag on when computed tokens will be produced. The time lag introduced by placing *k* initial tokens on the output port of actor A is $k*T_{AB}$. On the other hand, the amount of time needed by actor A to consume m tokens on its input port is $(m-1)*T_{CA}$. Hence, to ensure causality, we want to have:

$$k * T_{AB} \geq (m-1) * T_{CA}$$

$$\Rightarrow \quad k \geq \frac{(m-1) * T_{CA}}{T_{AB}}$$

$$\Rightarrow \quad k \geq \frac{(m-1) * n}{m}$$

This means that the minimum k to ensure causality is:

$$k_{min} = \left\lceil \frac{n\,(m-1)}{m} \right\rceil$$

It is interesting to note that this equation simplifies to k = 0 for homogeneous actors, which means initial tokens are not needed for homogeneous actors. In the Ptolemy II implementation of DT, we actually use the value $k = n$ for non-homogeneous actors, as a design simplification.

We conclude this appendix with an example. Consider the SDF graph shown in figure 43. We have $m = 3$, $n = 2$, $P = 1$, $R = 1$, $T_{CA} = 1/3$, and , $T_{AB} = 1/2$. We calculate $k = 2$. Hence we need at least two initial tokens on actor A.

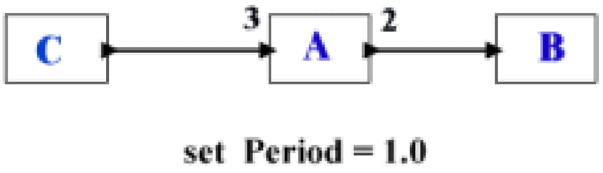


Figure 43: an example