Dynamic Dataflow Modeling in Ptolemy II

by Gang Zhou

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**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Edward A. Lee

Research Advisor

* * * * * * *

Professor Alberto Sangiovanni-Vincentelli

Second Reader

Published as:
Technical Memorandum UCB/ERL M05/2,
Electronics Research Laboratory,
University of California at Berkeley,
December 21, 2004.

Abstract

Dataflow process networks are a special case of Kahn process networks (PN). In dataflow process networks, each process consists of repeated firings of a dataflow actor, which defines a quantum of computation. Using this quantum avoids the complexities and context switching overhead of process suspension and resumption incurred in most implementations of Kahn process networks. Instead of context switching, dataflow process networks are executed by scheduling the actor firings. This scheduling can be done at compile time for synchronous dataflow (SDF) which is a particularly restricted case with the extremely useful property that deadlock and boundedness are decidable. However, for the most general dataflow, the scheduling has to be done at run time and questions about deadlock and boundedness cannot be statically answered. This report describes and implements a dynamic dataflow (DDF) scheduling algorithm under Ptolemy II framework based on original work in Ptolemy Classic. The design of the algorithm is guided by several criteria that have implications in practical implementation. We compared the performance of SDF, DDF and PN. We also discussed composing DDF with other models of computation (MoC). Due to Turing-completeness of DDF, it is not easy to define a meaningful iteration for a DDF submodel when it is embedded inside another MoC. We provide a suite of mechanisms that will facilitate this process. We give several application examples to show how conditionals, data-dependent iterations, recursion and other dynamic constructs can be modeled in the DDF domain.

Acknowledgements

I would like to thank my research advisor, Professor Edward A. Lee, not only for his encouragement and support during my most difficult time, but also for the pleasant group atmosphere and outstanding research environment he created. I am deeply inspired by his enthusiasm and devotion to pursuing excellent research.

I would also like to thank Professor Alberto Sangiovanni-Vincentelli for kindly agreeing to serve as the second reader of this report and providing valuable comments.

I would like to thank all of my colleagues in the Ptolemy group for creating such an excellent software environment and for all the stimulating discussions. Although I am an audience much of the time, I enjoyed every minute of it.

Finally, I would like to thank my wife Yan and my parents for their continuous love and encouragement. This report is dedicated to them.

Contents

1 Introduction

1.1 Concurrent Model of Computation (MoC) for Embedded System

One major achievement of traditional computer science is systematically abstracting away the physical world. The Von Neumann model provides a universal abstraction for sequential computation. The concept is so simple and powerful for transformational systems (vs. reactive systems [1][2]) that any program in traditional sense can run regardless of underlying platform ---- whether it is a supercomputer or a desktop. Embedded software systems, however, engage the physical world. Time, concurrency, liveness, robustness, continuums, reactivity, and resource management must be remarried to computation [1]. The traditional way to program embedded systems is to use assembly language or C, make sure the function implemented is correct, and then tweak some program parameters or priorities on the underlying RTOS and pray it meets the deadline imposed by the physical world. Important properties like time and concurrency are not part of the program model, but are only treated as afterthought when specifications are not met. This mismatch between the abstraction provided by the program model and the physical world results in programs that are hard to understand, difficult to debug and error-prone. Instead of the Von Neumann model for sequential computation, we need models of computation that support the concurrency inherent in embedded systems. However, since a universal model of computation for concurrent computation has yet to emerge, we need to understand how different models of computation deal with time and concurrency, use the ones which match the application domain well, and combine them in a systematic way.

One of the main objectives of the Ptolemy project is to experiment with different models of computation (called domains). The domains currently implemented in Ptolemy II include: Component Interaction (CI), Communicating Sequential Process (CSP), Continuous Time (CT), Distributed Discrete Event (DDE), Dynamic Dataflow (DDF), Discrete Event (DE), Discrete Time (DT), Finite State Machines (FSM), Giotto, Graphics (GR), Heterochronous Dataflow (HDF), Process Networks (PN), Parameterized Synchronous Dataflow (PSDF), Synchronous Dataflow (SDF), Synchronous Reactive (SR), and Timed Multitasking (TM). An essential difference between concurrent models of computation is their modeling of time [1]. Some are very explicit by taking time to be a real number that advances uniformly, and placing events on a time line or evolving continuous signals along the time line (such as CT, DE). Others are more abstract and take time to be discrete (such as DT, SR). Others are still more abstract and take time to be merely a constraint imposed by causality (such as PN, CSP, SDF, DDF). The last interpretation results in time that is partially ordered, which provides a mathematical framework for formally analyzing and comparing models of computation [3].

This report is mainly about one particular model of computation ----- dynamic dataflow (DDF). It describes and implements a DDF scheduling algorithm under the Ptolemy II framework. The following gives an outline of this report. In the remainder of section 1, we further motivate the use of dataflow MoC and briefly describe the Ptolemy II software environment upon which the DDF domain is created. Section 2 gives a review of denotational semantics of Kahn process networks, which is the theoretical foundation of any dataflow process network. Since Kahn process networks lack the notion of firing, we outline the treatment by Lee [8], which defines a continuous Kahn process as the least

fixed point of an appropriately constructed functional, which itself is defined by a firing function of a dataflow actor. We also give a brief survey of various dataflow models of computation. Section 3 is the main subject of this report. It describes the implementation of a DDF scheduling algorithm under the Ptolemy II framework. The design of the algorithm is guided by several criteria that have practical implications. We discussed designing actors used in DDF domain. We compared the performance of SDF, DDF and PN. We also discussed composing DDF with other models of computation (MoC). Due to Turing-completeness of DDF, it is not easy to define a meaningful iteration for a DDF submodel when it is embedded inside another MoC. We provided a suite of mechanisms that will facilitate this process. Section 4 gives several examples to show how conditionals, data-dependent iterations, recursions and other dynamic constructs can be modeled in DDF domain. Section 5 concludes the report and outlines some future directions.

1.2 Motivation for Dataflow Computing

The Dataflow model of computation is most popularly used in digital signal processing (DSP). Dataflow programs for signal processing are often described as directed graphs where each node represents a function and each arc represents a signal path. Compared with imperative programs which do not often exhibit the concurrency available in the algorithm, dataflow programs automatically break signal processing tasks into subtasks and thus expose the inherent concurrency as a natural consequence of programming methodology. In a dataflow graph, each function node executes concurrently conceptually with the only constraint imposed by data availability. Therefore it greatly facilitates efficient use of concurrent resources in the implementation phase.

Dataflow programming has come a long way since its use in the signal processing community. Various programming environments based on this model of computation have been developed over the years. Various specialized dataflow models of computation have been invented and studied. (We will give a brief overview in section 2). Various scheduling algorithms based on different criteria pertinent for specific applications have been proposed. It is still a fruitful area of active research due to its expressive way to represent concurrency for embedded system design.

1.3 A Software Environment for Experimenting with MoC ---- Ptolemy II

The Ptolemy project at University of California at Berkeley studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interaction between components [4]. Unlike most system design environments which have one or two built-in models of computation, the Ptolemy kernel has no built-in semantics. Instead the kernel package defines a small set of Java classes that implement a data structure supporting a general form of uninterpreted clustered graphs (see Figure 1.1 taken from [6]), which provide an abstract syntax that is not concerned with the meaning of the interconnections of components, nor even what a component is.

The semantics is introduced in the actor package which provides basic support for executable entities. However it makes a minimal commitment to the semantics of these entities by avoiding specifying the order in which actors execute and the communication mechanism between actors. These properties are defined in each domain where the

director controls the execution of actors and the receiver controls the communication between actors. Figure 1.2 shows a graphical user interface in Ptolemy II called Vergil where we build models most of time.

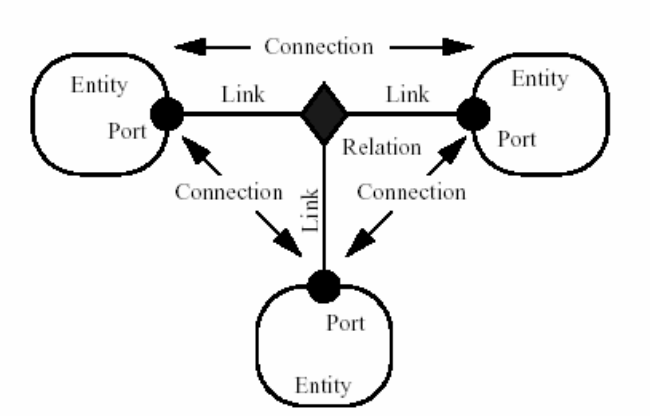


Figure 1.1 Abstract syntax of Ptolemy II

There is a large actor library in Ptolemy II. Some actors are written to be domain-polymorphic, i.e., they can operate in any of a number of domains. They are the result of principle of separation of function and communication. Some actors can only be used in specific domains because they deal with functionalities specific to those domains. It is usually much more difficult to write domain-polymorphic actors.

There are three big volumes of design documents which cover details of the software architecture [5][6][7]. They are usually updated with each software release.

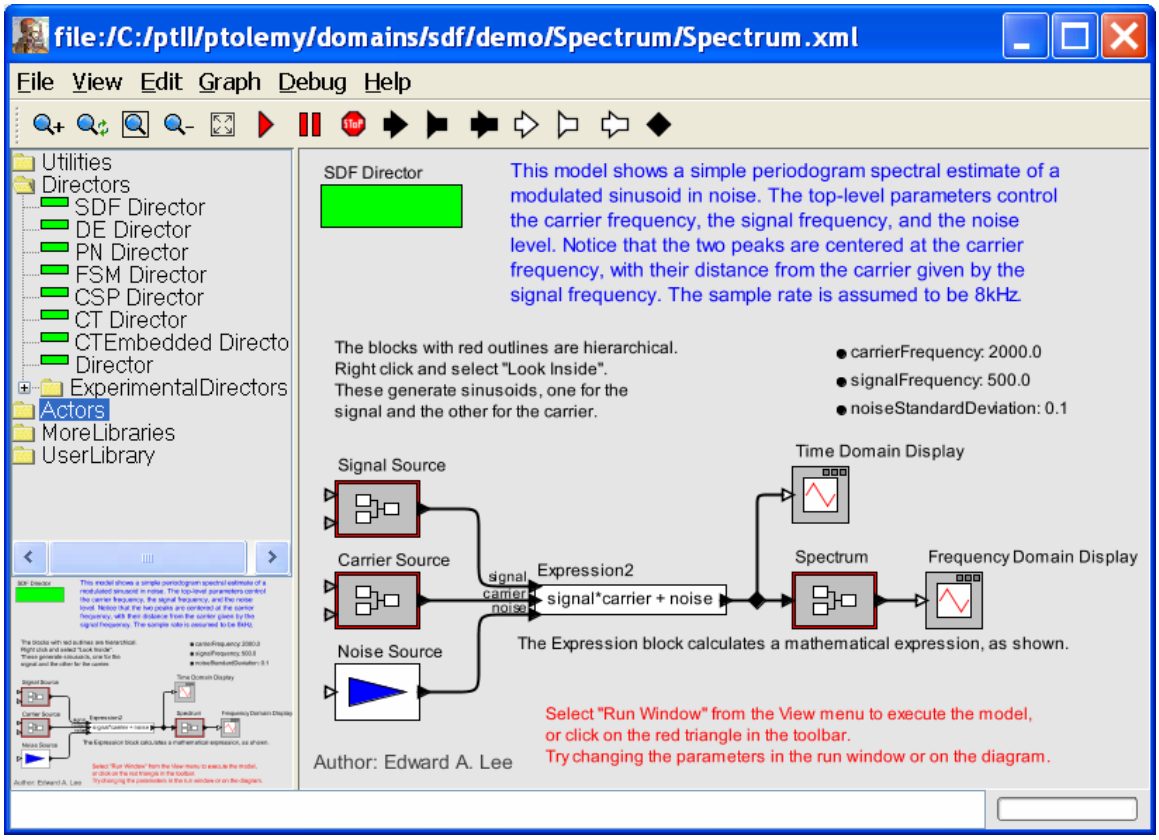


Figure 1.2 Vergil --- a graphical user interface for Ptolemy II

2 Semantics of Dataflow Process Network

Dataflow can be viewed as a special case of Kahn process networks. Therefore in section 2.1 we give a brief overview of denotational semantics of Kahn process networks to serve as the theoretical foundation for the dynamic scheduler we report here. Kahn process networks lack the notion of firing, therefore in section 2.2 we outline the treatment by Lee [8], which defines a continuous Kahn process as the least fixed point of an appropriately constructed functional which itself is defined by the firing function of a dataflow actor. In section 2.3 we give a brief survey of various dataflow models of computation, serving to show where DDF domain stands in the whole spectrum. The contents of section 2 are mostly based on [8][9]. Interested readers should refer to them for more details.

2.1 Fixed Point Semantics of Kahn Process Network

A Kahn process network [10] is a network of processes that communicate only through unidirectional, single input single output FIFO channels with unbounded capacities. Each channel carries a possibly infinite sequence that we denote $s = [v_1, v_2, v_3 \ldots]$, where each v_i is an atomic data object called a token drawn from a set V. Each token is written (produced) exactly once, and read (consumed) exactly once. So it has event semantics instead of state semantics as in some other domains such as Continuous Time. The set of all such sequences (finite and infinite) is denoted S. The set of tuples of n such sequences is denoted S^n. A Kahn process is a mapping F: $S^m \rightarrow S^n$ from an m-tuple to an n-tuple, with a key technical restriction that the mapping must be a continuous function (to be defined).

A partially-ordered set (poset) is a set S where an ordering relation (denoted $\sqsubseteq$) is defined that is reflexive ($\forall s \in S, s \sqsubseteq s$), transitive ($\forall s, s', s'' \in S, s \sqsubseteq s', s' \sqsubseteq s'' \Rightarrow s \sqsubseteq s''$) and anti-symmetric ($\forall s, s' \in S, s \sqsubseteq s', s' \sqsubseteq s \Rightarrow s = s'$). The bottom element of a poset S, if it exists, is an element $s \in S$ that satisfies $\forall s' \in S, s \sqsubseteq s'$. An upper bound of a subset $W \subseteq S$ is an element $s \in S$ that satisfies $\forall w \in W, w \sqsubseteq s$. A least upper bound (LUB) of a subset $W \subseteq S$ is an upper bound s such that for any other upper bound s', $s \sqsubseteq s'$. A chain in S is a totally ordered subset of S, i.e., any two elements in the subset are ordered. A complete partial order (CPO) is a poset with a bottom element where every chain has a LUB.

There is a natural ordering relation in the set S consisting of all finite and infinite sequences. It is the prefix order, defined as $\forall s, s' \in S, s \sqsubseteq s'$ if s is a prefix of s'. This definition easily generalizes to S^n element-wise. This turns out to be a very useful ordering relation in defining the denotational semantics of Kahn process network because under this ordering S^n is a CPO and there is strong theorem for continuous functions defined on a CPO.

A function F: $S^m \rightarrow S^n$ is monotonic if $\forall s, s' \in S^m, s \sqsubseteq s' \Rightarrow F(s) \sqsubseteq F(s')$. This expresses an untimed notion of causality since according to the definition, giving additional inputs can only result in additional outputs. A function F: $S^m \rightarrow S^n$ is continuous if for every chain $W \subseteq S^m$, $F(W)$ has a LUB and $F(LUB(W)) = LUB(F(W))$. Here $F(W)$ denotes a set obtained by applying the function F to each element of W. Intuitively continuity means the response of the function to an infinite input sequence is the limit of its

response to the finite approximations of this input. It is easy to prove a continuous function is always monotonic, but not vice versa.

Finite composition of Kahn processes is obviously determinate (i.e, given the input sequences, all other sequences are determined) if there is no feedback. With feedback there may be zero, one or multiple behaviors. An interpretation due to Kahn called the least-fixed-point semantics is now the well adopted denotational semantics for Kahn process networks. It is due to a well known fixed point theorem that states that a continuous function $F: S^n \to S^n$ in CPO S^n has a least fixed point s (i.e., $F(s) = s$ and for any other s' satisfying $F(s') = s'$, $s \sqsubseteq s'$). In addition, it gives a constructive way to find the least fixed point, which is the LUB of the following sequence: $s_0 = \Lambda, s_1 = F(s_0), s_2 = F(s_1)...$ where Λ is the tuple of empty sequences. Notice that this way of finding least fixed point matches exactly with operational semantics if all signals start with empty sequences. Thus we are assured our scheduling algorithm will generate sequences that are guaranteed to be the prefix of denotational semantics.

2.2 Denotational Semantics for Dataflow with Firing

So far these process networks fail to capture an essential principle of dataflow, proposed by Dennis and used in almost all practical implementations of dataflow, that of an actor firing. An actor firing is an indivisible quantum of computation. A set of firing rules give preconditions for a firing, and the firing consumes tokens from the input streams and produces tokens on the output streams. In [8] Lee shows that sequences of firings define a continuous Kahn process as the least fixed point of an appropriately constructed

functional, therefore formally establishing dataflow process network as a special case of Kahn process network. We give a brief review of that paper for completeness.

A dataflow actor with *m* inputs and *n* outputs is a pair $\{f, R\}$, where:

1. f: $S^m \rightarrow S^n$ is a function called the firing function,

2. $R \subseteq S^m$ is a set of finite sequences called the firing rules,

3. f(r) is finite for all $r \in R$, and

4. no two distinct $r, r' \in R$ are joinable (i.e. they don't have a LUB).

A Kahn process F based on the dataflow actor $\{f, R\}$ can be defined as follows:

$$F(s) = \begin{cases} f(r).F(s') \text{ if there exists } r \in R \text{ such that } s = r.s' \\ \Lambda \text{ otherwise} \end{cases}$$

where **.** represents concatenation. Since F is self-referential in this definition, we can not guarantee that F exists, nor that F is unique. To show its existence and uniqueness, it has to be interpreted as the least-fixed-point function of functional $\phi : (S^m \rightarrow S^n) \rightarrow (S^m \rightarrow S^n)$ defined as

$$(\phi(F))(s) = \begin{cases} f(r).F(s') \text{ if there exists } r \in R \text{ such that } s = r.s' \\ \Lambda \text{ otherwise} \end{cases}$$

where $(S^m \rightarrow S^n)$ represents the set of functions mapping S^m to S^n. It can be proved that $(S^m \rightarrow S^n)$ is a CPO under pointwise prefix order. Another similar notation $[S^m \rightarrow S^n]$ represents the set of continuous functions mapping S^m to S^n, and it can also be proved to be a CPO under pointwise prefix order. Obviously $[S^m \rightarrow S^n] \subset (S^m \rightarrow S^n)$.

It can be proved that ϕ is both monotonic and continuous (see [8]). Again using the same theorem, ϕ has a least fixed point (which is a function in this case) and there is a

constructive procedure to find this fixed point. Starting with the bottom of the poset $(S^m \to S^n)$, $\psi : S^m \to S^n$, which returns Λ, an *n*-tuple of empty sequences, we can construct a sequence of functions: $F_0 = \psi, F_1 = \phi(F_0), F_2 = \phi(F_1)...$, and the LUB of this chain is the least fixed point of ϕ. If we apply this chain of functions to $s \in S^m$ where $s = r_1.r_2.r_3...$ and $r_1, r_2, r_3... \in R$, we will get $F_0(s) = \Lambda, F_1(s) = f(r_1), F_2(s) = f(r_1).f(r_2),....$ This exactly describes the operational semantics of Dennis dataflow for a single actor. Since each $F_i \in [S^m \to S^n]$ is a continuous function and the set $[S^m \to S^n]$ is a CPO as mentioned previously, the LUB of the chain is continuous and hence describes a valid Kahn process that guarantees determinacy. Notice that the firing function *f* need not be continuous. In fact, it does not even need to be monotonic. It merely needs to be a function, and its value must be finite for each of the firing rules.

The conditions satisfied by firing rules is extended in [8] to describe composition of dataflow actors such as two-input, two-output identity function. Readers should refer to the paper for further details.

2.3 A Review of Dataflow MoC

----- from Static Analysis to Dynamic Scheduling and In-Between

Over the years, a number of dataflow models of computation (MoC) have been proposed and studied. In the synchronous dataflow (SDF) MoC studied by Lee and Messerschmitt [11], each dataflow actor consumes and produces fixed number of tokens on each port in each firing. The consequence is that the execution order of actors can be statically determined prior to execution. This results in execution with minimal overhead, as well

as bounded memory usage and a guarantee that deadlock will never occur. The cyclo-static dataflow MoC[12] extends SDF by allowing each actor's consumption and production rates to vary in a cyclic but predetermined pattern. It does not add more expressiveness to SDF, but turns out to be more convenient to use in some scenarios. The heterochronous dataflow MoC (HDF) proposed by Girault, Lee and Lee [13] extends SDF by using so-called modal model to compose FSM with SDF. It allows actors to change their rate signatures between global iterations of a model. In case of a finite number of rate signatures, properties like consistency and deadlock are still decidable. Bhattacharya and Bhattacharya proposed a parameterized synchronous dataflow (PSDF) MoC [14], which is useful for modeling dataflow systems with reconfiguration. In this domain symbolic analysis of the model is used to generate a quasi-static schedule that statically determines an execution order of actors, but dynamically determines the number of times each actor fires. Buck proposed a Boolean dataflow (BDF) MoC [15] which allows the use of some dynamic actors such as BooleanSelect and BooleanSwitch. He extends the static analysis techniques used in SDF and in some situations a quasi-static schedule can be pre-computed. But fundamentally since BDF is Turing-complete, it does not guarantee that the scheduling algorithm will always succeed. If it fails, a dynamic dataflow (DDF) MoC [16] should be used to execute the model. DDF uses only runtime analysis and thus makes no attempt to statically answer questions about deadlock and boundedness. Since this section only serves as a brief review, it suffices to say that the list here does not include every variant of dataflow ever conceived.

3 Design and Implementation of a DDF Domain

This section is the core of this report. It describes and implements a DDF scheduling algorithm under the Ptolemy II framework. In section 3.1, several criteria are used to guide the design of the DDF scheduling algorithm. Section 3.2 continues with the implementation of the algorithm. In section 3.3, we use several actors such as BooleanSelect and Select to show how dynamic actors are designed to be used in DDF domain. Section 3.4 gives performance comparison of SDF, DDF and PN domains. In Section 3.5, we discuss composing DDF with other MoCs.

3.1 Criteria and algorithm for DDF Scheduling

Given a DDF graph, there can be numerous ways to schedule its execution. The simplest way one can think of is to keep firing any arbitrary enabled actor until either the graph is deadlocked or some predefined stop condition is reached. However, there are many problems with this naïve scheduler. For one, the execution may need unbounded memory whereas some other scheduler may only need bounded memory. Therefore we need some criteria to determine whether some scheduler is a "good" scheduler. It is important to point out that there are no absolute criteria. Different people can pose different criteria as long as they can interpret their choice in a reasonable way.

The criteria we used were first realized in Ptolemy Classic [17], where the first criterion has higher priority over the second one, the second one over the third one and so on:

1. After any finite time every signal is a prefix of the LUB signal given by the denotational semantics. (Correctness)

2. The scheduler should be able to execute a graph forever if it is possible to execute a graph forever. In particular, it should not stop prematurely if there are enabled actors. (Liveness)
3. The scheduler should be able to execute a graph forever in bounded memory if it is possible to execute the graph forever in bounded memory. (Boundedness)
4. The scheduler should execute the graph in a sequence of well-defined and determinate iterations so that the user can control the length of an execution by specifying the number of iterations to execute. (Determinacy)

If a schedule satisfies condition 1, it is called a correct execution. To get a correct execution, we require that the operational semantics of a graph defined by a scheduler be consistent with its denotational semantics. Note that this does not require each signal to converge to LUB semantics. As pointed out by Edward Lee, if leaving the execution at a finite prefix were incorrect, then it would be incorrect for Ptolemy II to stop the execution when a stop button is pushed. This would be counterintuitive. In practice, any execution is always partial with respect to denotational semantics with infinite length signals.

If a schedule satisfies condition 2 and 3, it is called a useful execution. In particular, condition 2 requires that for every non-terminating model, after any finite time, the execution will extend at least one signal in additional finite time. Note the subtlety here. It does not require to extend every signal. Condition 3 is for the purpose of practical implementation. Among all possible schedulers, we would prefer those that can execute the graph with bounded memory. As pointed out by Parks [16], liveness is a property of the model itself (directly related with lengths of signals in its denotational semantics),

whereas boundedness is a property of both the model and its scheduler. Buck proved in his Ph.D. thesis [15] that a BDF graph is Turing-complete. Since the BDF domain is a subset of the DDF domain, a DDF graph is also Turing-complete. This has the consequence that both liveness and boundedness are undecidable for DDF graphs, meaning that no finite analysis can answer the question about these properties for any arbitrary DDF model. However, since a DDF scheduler has infinite time to run an infinite execution, there may exist schedulers that can satisfy both conditions, as Parks' algorithm did for Process Networks.

Condition 4 expresses the desire to extend the concept of iterations in the SDF domain to DDF domain. In SDF, one iteration is defined as the minimum (but non-empty) complete cycle through which the graph returns to its original state (defined as the number of tokens in each arc). This can be determined by doing static analysis on the SDF graph. Then an SDF graph can be executed by repeating iterations, and the user can control the length of execution by specifying the number of iterations in one execution. In DDF, there is no such inherent iteration. But we still want to define a somewhat meaningful iteration which achieves something such as printing a dot on a plotter (which involves invoking that actor once). And the set of actors executed in each iteration is determinate so that the state of the model is a well-defined function of the number of iterations. This requires that which actors to fire in each iteration should not depend on arbitrary decisions made by the scheduler, like the order in which it examines the enabling status of each actor. This way the user can control the progress of the model using the same mechanism as in SDF. The detailed implementation will be explained in the later sections.

It is appropriate to repeat that the set of criteria we give are subjective but based on good reasoning. For example, Marc Geilen and Twan Basten proposed a different set of criteria in a recent paper [18]. They replaced condition 2 with an output completeness condition, i.e., each signal should eventually converge to that prescribed by the denotational semantics. They observed that Parks' algorithm [16] cannot solve so-called local deadlock problem where several actors are deadlocked in a cyclic fashion but the whole model can still make progress, therefore Parks' algorithm won't try to solve these deadlocks since it will only do so when the whole model is deadlocked. Thus the execution will violate their completeness condition. Since they cannot come up with a scheduler that would satisfy their criteria for all models, their solution is to give a executing strategy that would satisfy their criteria for a subset of models, which are bounded and effective. (Their definition for effectiveness is that every token produced will be ultimately consumed.) However, due to Turing-completeness, whether or not a model is bounded and effective is undecidable. Therefore there is no program which can classify any arbitrary model before execution. And if we prefer completeness over boundedness, some models may run out of memory, whereas using our criterion they may be executed forever in bounded memory. We prefer the our way because it allows useful functions provided by some part of the model to be performed in bounded memory whereas with Geilen & Basten's technique the model may have to be aborted at some point due to memory constraints. For example, the model in Figure 3.1 can be executed as the model in Figure 3.2 with bounded memory using our criterion. If we insist on complete execution, then we will run out of the memory eventually. So this is a something-better-than-nothing philosophy.

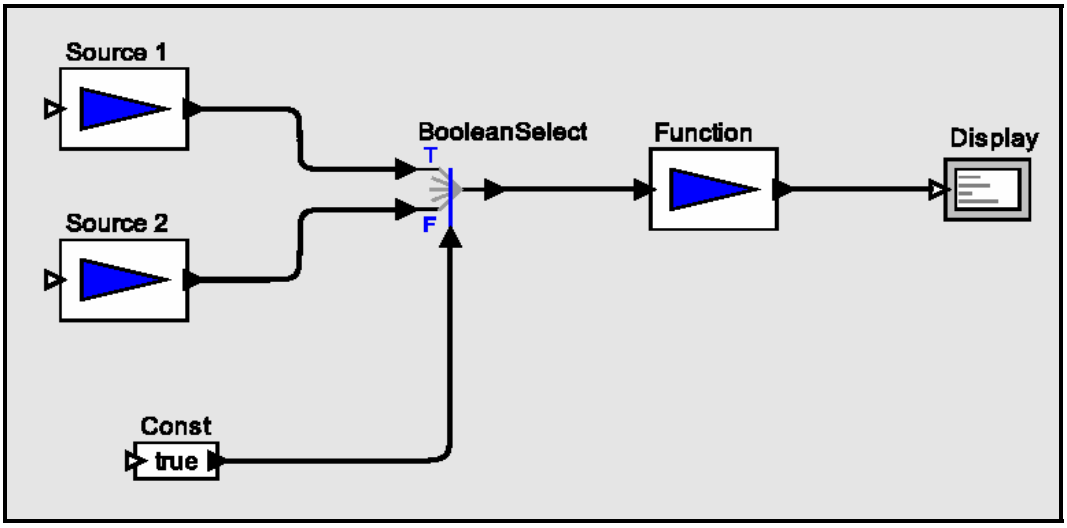


Figure 3.1 A dataflow model where BooleanSelect actor only consumes from true input port

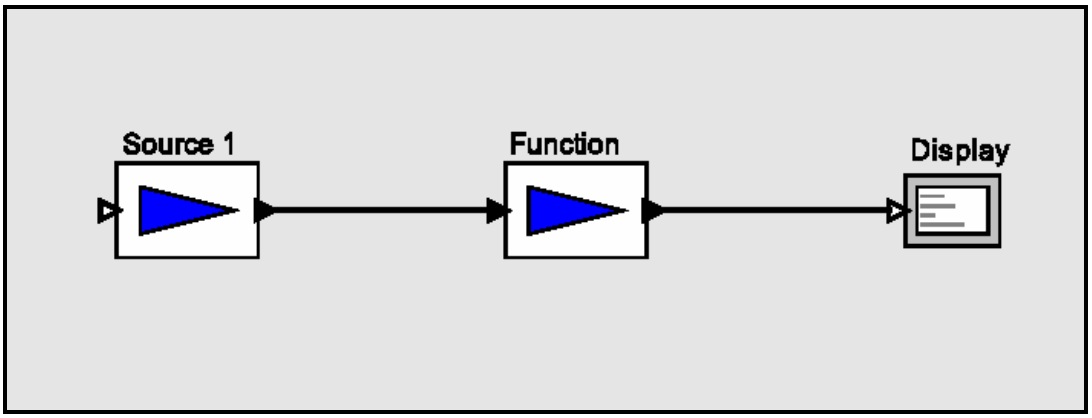


Figure 3.2 A simplified model for Figure 3.1 from the viewpoint of Display actor

The particular scheduler implemented in this DDF domain is based on a scheduler in Ptolemy Classic [17]. The intuition behind this scheduler is that we defer firing of each enabled actor until it is absolutely necessary in order to avoid deadlock. Thus we don't create more tokens than necessary on each channel. To capture this notion, we define a **deferrable** actor as an actor, one or more of whose output channels already has enough tokens to satisfy the demand of the destination actor. There are several points to notice here. First, if the actor has multiple output channels, it only takes one output channel satisfying one destination actor connected to that channel to be a deferrable actor. Of

course if there are more than one output channel satisfying destination actors, the actor is still deferrable. However, it does not require that all output channels satisfy all destinations actors to be a deferrable actor. Second, when one particular channel of a destination actor is satisfied, it has two possible situations. One situation is that the number of tokens buffered on that channel is greater than or equal to the number of tokens required on that channel in order to fire the destination actor. Another situation is that the destination actor does not consume a token on that channel during next firing. Third, when one particular channel of a destination actor is satisfied, it does not mean the destination actor is enabled. It may be still waiting for tokens on different channels to enable the next firing. The consequence is that we cannot use the *prefire()* method of the destination actors to check its deferability. Each actor must expose the number of tokens consumed on each input channel as part of its interface, which may dynamically change from one firing to another. This will have consequences in designing actors that can be used in DDF domain.

Having defined the deferability of an actor, which is a key concept in our scheduler, we now give the algorithm in the following pseudo-code:

```
At the start of each basic iteration compute {
        E = set of enabled actors
        D = set of deferrable and enabled actors
        minimax(D) = subset of D as defined on the next page
}
One basic (default) iteration consists of {
        If (E \ D != Ø)
                fire (E \ D)
        else if (D != Ø)
                fire minimax(D)
        else
                declare deadlock
}
```

where the function "minimax(D)" returns a subset of D with the smallest maximum number of tokens on their output channels which satisfy the demands of destination actors, and the backslash denotes set substraction.

Let's check this algorithm against the four criteria we proposed at the beginning of this section. We fire each actor only when it is enabled, and we assume each actor is written such that at the beginning of each firing, it will consume certain number of tokens corresponding to one of its firing rules from each input channel; then it accurately performs its computation (in other words the code in each actor has no bug); finally it produces tokens resulting from tokens it just consumed to the output channels. Therefore the operational semantics should be consistent with the denotational semantics, and we have a correct execution.

It is easy to see that the only time the scheduler will declare a deadlock is when both if-clauses return false, i.e.,

$$E \setminus D == \emptyset \;\&\&\; D == \emptyset.$$

This is equivalent to

$$E == D == \emptyset.$$

Therefore we can conclude that the scheduler will declare deadlock only when there are no enabled actors (E == Ø), i.e., when all actors are read-blocked. This guarantees that the model will be executed forever if it can be executed forever. In particular, it won't stop prematurely if there are still enabled actors.

Parks designed an algorithm which guarantees bounded execution for a process network if it can be executed with bounded memory. The basic idea is to start with a small buffer size on each channel, which will introduce a write-block if the output channel is full. Then the model is executed until a global deadlock is reached. At this time, if some actors are write-blocked (i.e., it is an artificial deadlock due to the limited buffer size), increase the smallest **full** buffer size among those channels where write-block occurs. Then continue to execute the model and repeat the deadlock-resolving mechanism if necessary. This technique cannot be directly applied to dataflow scheduling because the firing of a dataflow actor is atomic. Once it is initialized, the firing cannot be suspended until all output tokens are produced. For example, in Figure 3.3, the source actor A produces 10 tokens in each firing and the sink actor B consumes 1 token in each firing. In the PN (process network) domain, this model can be executed with buffer size of 1 because the process associated with actor A will suspend itself if the output channel has already been filled with one token and continue to produce the next token only after the process associated with actor B has consumed the token and emptied the channel. If the same model is executed under a dataflow scheduler, when actor A fires, it will require a buffer size of 10 on the output channel to store all tokens produced in each firing. Therefore whatever dataflow scheduler is used, a buffer size of at least 10 is required to execute this model.

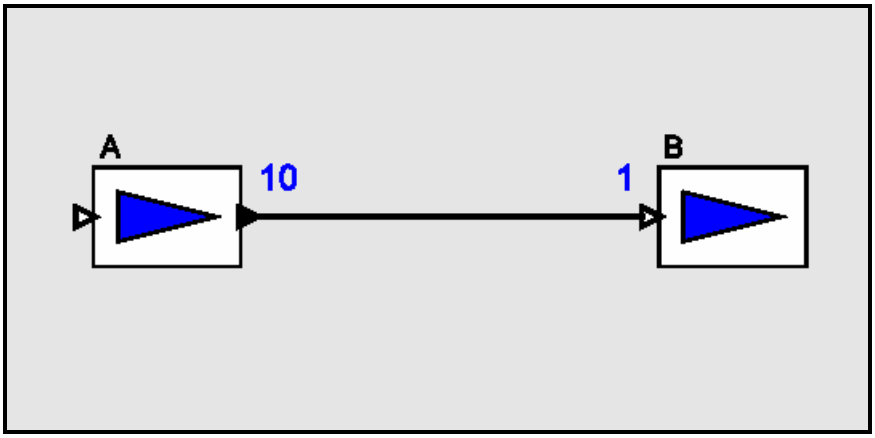


Figure 3.3 A simple model with two actors

In order to preserve bounded memory whenever possible, a dataflow actor should not produce more tokens to the output channels when there are already enough tokens to satisfy the demand of destination actors. Therefore during each basic iteration we only fire enabled and non-deferrable actors (the set E \ D) if possible. Assume each actor consumes or produces at most N tokens in each channel during each firing. There could be as many as N-1 tokens on a channel with the demand still unsatisfied. An actor could produce as many as N tokens on each output channel when fired, so there could be as many as $2N$-1 tokens on a channel. If deferrable actors are never fired, then there will never be more than $2N$-1 tokens on any channel. However, if at some point during the execution, all enabled actors become deferrable (E == D), we will fire minimax(D) according to the algorithm. The motivation is to increase the capacity of only the smallest full channels as Parks' algorithm did for process networks. Each deferrable actor has at least one output channel with a satisfied demand. In some sense, such channels are full. In general we do not know which channels an actor firing will produce tokens on, so for each deferrable actor we must determine the largest such channel buffer size, which is denoted as N_i for actor i. We choose to fire actors with the smallest value for N_i (the set minimax(D)). Some arguments presented here are vague, and as of this writing we don't

have a rigorous proof yet that the algorithm satisfies criterion 3. It remains further work. However, our experience points to that direction and indeed it is a challenge to come up with a model that can be executed in bounded memory but fails to do so with our algorithm.

With the way the algorithm is designed, the set of actors that are executed in each basic iteration is determinate. The reason is that only at the start of each basic iteration are all actors classified into sets E, D, minimax(D). These sets are not updated in the middle of one basic iteration. Each set is a function of a model's state represented by tokens (their number and values) queued on each channel. By "function" we mean the contents of each set do not depend on the order in which we examine the status of each actor. The resulting state after firing a set of actors does not depend on the order in which we fire each actor in the set. Therefore we can conclude that each basic iteration is well-defined and determinate. Later on we will show the mechanism to group several basic iterations into one iteration that is more meaningful.

3.2 Implementation of DDF Domain in Ptolemy II

This section continues with the implementation of a scheduler under the Ptolemy II framework. In Ptolemy II, each domain has a director which is responsible for controlling the execution of actors in that domain. In some domains, such as CT, SDF and SR, the model can be statically analyzed (called compiling a model) to come up with a schedule before execution starts. That is the job of a scheduler. In other domains such as DE, firings of actors can only be dynamically scheduled during the execution. For these domains, there is no (static) scheduler per se and all functionality (including dynamic

scheduling and actor invocation since they are closely intertwined) is implemented in the director. The DDF domain belongs to the latter case and therefore a single DDFDirector does dynamic scheduling as well as actor invocation.

In Ptolemy II, the semantics of a domain is defined both by the director and the receiver. The director is responsible for controlling the flow of execution and the receiver is responsible for mediating communication between actors. Since the communication style in a dataflow network is asynchronous message passing, we need a first-in-first-out queue to act as the receiver in the DDF domain. There are two kinds of receivers that are already implemented in the software which can be used for our purpose. One is the SDFReceiver, which contains a FIFOQueue implemented with an array. Another is the QueueReceiver which contains a FIFOQueue implemented with a LinkedList. We choose to use the SDFReceiver in this implementation.

According to the execution model in Ptolemy II, a director controls how actors are invoked with the action methods defined in the Executable interface. A brief description of the implementation for each method is given here.

preinitialize()

For a static scheduling director, this is the place where a static schedule is computed. Since DDFDirector does dynamic scheduling, we simply inherit this method from the base class.

initialize()

In this method, we classify all actors into three categories: actors that are not enabled, actors that are enabled but deferrable, and actors that are enabled and not deferrable. We also search those actors for a parameter named *requiredFiringsPerIteration*, which specifies the number of times the actor must be fired in one iteration. Since some actors may not get fired in one basic iteration, one iteration consists of several basic iterations (possible infinite if the model is ill-designed). The reason we introduce this mechanism to define an iteration is to match the user's expectation. For example, intuitively a user would expect to see one token consumed and plotted by a sink actor in each iteration, which can be achieved by adding the parameter to the sink plotter with value 1.

prefire()

If the DDF domain is embedded in another domain, we first check the input ports of the container composite actor to see whether they have enough tokens. It is tricky to do that because unlike SDF, it is undecidable for DDF to determine the exact number of tokens required to finish one iteration. This point will be further elaborated in section 3.5 when we mix DDF domain with other Ptolemy II domains. We also reset to zero the counting variables for those actors which are required to fire specified number of times in each iteration.

fire()

This is the place where the bulk of computation is performed. Each invocation of this method corresponds to one iteration of the model, which by default is one basic iteration. However, if some actor has a parameter named *requiredFiringsPerIteration*, continue to execute basic iterations until the actor has been fired at least the number of times given

by that parameter. If more than one actor has such a parameter, then the iteration will continue until all are satisfied. As an optimization technique, after each actor is fired, we determine the new status (is it enabled? is it deferrable?) of the actors directly connected to this actor as well as itself since the rest of the actors won't be affected. Then we don't need to do classification for each actor at the beginning of the next iteration.

posfire()

This method returns false if it is determined that the execution of the model should not continue to the next iteration. This happens when the number of iterations specified in the director parameter has been reached (there is no upper limit if the parameter is left with its default value 0) or the model comes to a deadlock in the current iteration.

wrapup()

This method is invoked exactly once at the end of the execution for cleaning up purposes. We simply inherit it from the base class.

3.3 Design of Actors Used in DDF Domain

Given the scheduling algorithm in previous sections, we now consider the implementation details such as how to determine whether an actor is enabled. What kind interface should an actor expose so that the director can do proper scheduling based on the information provide by the actor's interface? We will generalize the mechanism used in the SDF domain.

In SDF, the scheduling algorithm relies on the actors in the model to declare the data rates of each port. The data rates of ports are specified using up to three parameters on each port named *tokenConsumptionRate*, *tokenProductionRate*, and *tokenInitProduction*. The production parameters are valid only for output ports, while the consumption parameter is valid only for input ports. If a valid parameter is not specified when the scheduler runs, then default values of the parameters corresponding to a homogeneous actor will be assumed: input ports are assumed to have a consumption rate of one, output ports are assumed to have a production rate of one, and no tokens produced during initialization. If there are multiple channels connected to a multiport, all channels are assumed to have the same rate given by the parameter.

In DDF, the director also relies on the *tokenConsumptionRate* parameter of each input port to determine whether an actor is enabled or deferrable. (The *tokenProductionRate* parameter has no use in DDF because for a general DDF actor this information is not available before the actor gets fired. The *tokenInitProduction* serves the same purpose as in SDF, which is the initial token production rate during the initialization phase, used to generate initial tokens for the model at the beginning of the execution.) However, unlike an SDF actor, the dynamic nature of DDF actor dictates that the parameter will have a possible new value after each firing and thus must be updated if needed.

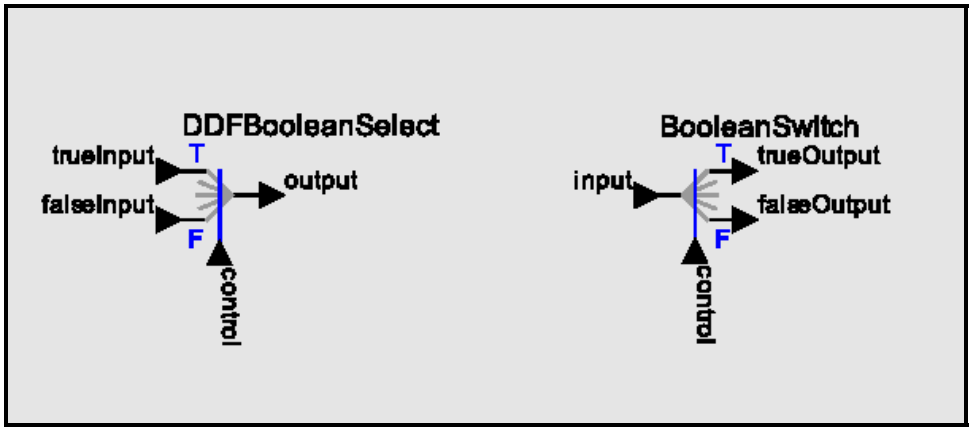


Figure 3.4 Icons for DDFBooleanSelect and BooleanSwitch actors

For example, the BooleanSelect Actor is a canonical actor in DDF domain and its design is representative of DDF actors. (See Figure 3.4, where it is renamed DDFBooleanSelect to distinguished it from BooleanSelect used in other domains such as DE. We are currently working on a domain polymorphic version.) It has the following firing rules:

$$\left\{ ([*],\ \perp,\ [T]),(\ \perp,\ [*],\ [F]\ ) \right\}$$

where the first rule says if the control port has a true token, the trueInput port must have a token and the second rule says if the control port has a false token, the falseInput port must have a token. Compared with SDF actor firing rule which only specifies the number of tokens each input port must consume, the firing rules of BooleanSelect actor also assert the token values of some input port. It seems that just providing the rate information wouldn't be enough to distinguish between different firing rules. However, it turns out that for a large category of actors which use so-called sequential firing rules, we can decompose original actor firing which would be enabled only when one of firing rules is satisfied into multiple firings and in each of new firings, the rate information would be enough to determine if an actor is enabled.

Before proceeding with how this can be achieved with BooleanSelect actor, let's spend a few words on sequential firing rules. Intuitively sequential means that the firing rules can be tested in a pre-defined order using only blocking reads. Each blocking read can be expressed as rate information on the input ports. Upon consuming the token, the actor determines the next input port to read token from, and the corresponding rate information is updated to reflect that. For a more rigorous treatment of this rather technical definition of sequentiality, please refer to [9]. Later on in this section, we will give an example of non-sequential firing rules.

The firing rules of BooleanSelect actor are sequential, therefore we can introduce a two-phase firings for this actor. During initialization, the control port sets its rate to 1 and trueInput/falseInput ports both set their rates to 0.

This code block from DDFBooleanSelect actor shows initialization:

```
public void initialize() throws IllegalActionException {
        super.initialize();
        _isControlRead = false;
        trueInput_tokenConsumptionRate.setToken(new IntToken(0));
        falseInput_tokenConsumptionRate.setToken(new IntToken(0));
        control_tokenConsumptionRate.setToken(new IntToken(1));
 }
```

In the first firing, the actor consumes a Boolean token from control port. Depending on the Boolean value of that token (true or false), the corresponding port (trueInput or falseInput) changes its rate to 1 and the other port keeps its rate at 0. The control port also needs to change its rate to 0 to declare that it doesn't consume token in the next firing. In the second firing, trueInput port or falseInput port with rate 1 consumes one token and

sends it to the output port. Then the actor resets the rate parameters of all input ports just as before first firing.

This code block from DDFBooleanSelect actor shows two-phase firings:

```
public void fire() throws IllegalActionException {
        if (_isControlRead) {
            if (_control) {
                output.send(0, trueInput.get(0));
            } else {
                output.send(0, falseInput.get(0));
            }
            _isControlRead = false;
        } else {
            _control = ((BooleanToken)control.get(0)).booleanValue();
            _isControlRead = true;
        }
}
```

This code block from DDFBooleanSelect actor shows the rate update after each firing:

```
public boolean postfire() throws IllegalActionException {
        if (_isControlRead) {
            if (_control) {
                trueInput_tokenConsumptionRate.setToken(new IntToken(1));
                falseInput_tokenConsumptionRate.setToken(new IntToken(0));
                control_tokenConsumptionRate.setToken(new IntToken(0));
            } else {
                trueInput_tokenConsumptionRate.setToken(new IntToken(0));
                falseInput_tokenConsumptionRate.setToken(new IntToken(1));
                control_tokenConsumptionRate.setToken(new IntToken(0));
            }
        } else {
            trueInput_tokenConsumptionRate.setToken(new IntToken(0));
            falseInput_tokenConsumptionRate.setToken(new IntToken(0));
            control_tokenConsumptionRate.setToken(new IntToken(1));
        }
        return super.postfire();
}
```

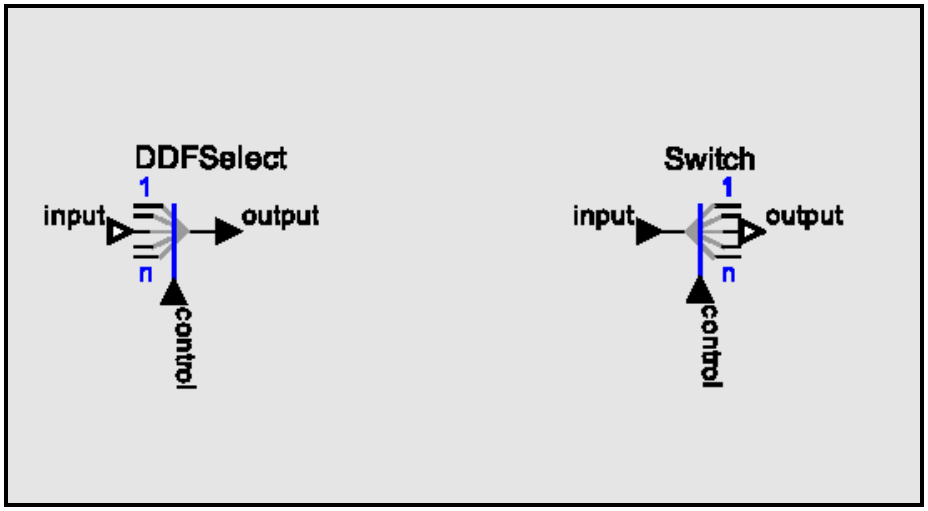


Figure 3.5 Icons for DDFSelect and Switch actors

Another extension of the SDF domain involves rate declaration for multiports, which allow multiple-channel connections. In SDF, all channels connected to the same multiport have the same rate. This won't work for some dynamic actors. For example, the Select actor is very similar to BooleanSelect actor functionally. (See Figure 3.5 where we use the name DDFSelect to distinguish from Select used in other domains such as DE.) Its input port is a multiport (represented by hollow arrow) because the actor communicates via an indeterminate number of channels depending on the connections made to the input port. The control port consumes an integer token, and its value specifies the input channel that should be read in the next firing. In this case, we use an ArrayToken for each multiport to represent the rates of channels. The length of the array is equal to the width of the port which is the number of channels connected to the port. Each element of the array represents the rate of a channel in the order channels are created while building the model. The rest of functionality including two-phase firings is the same as for the DDFBooleanSelect Actor.

It is interesting to notice that two actors BooleanSwitch and Switch (see Figure 3.4 and 3.5), the counterparts of BooleanSelect and Select, don't need special treatment to be used in DDF domain. They only have one firing rule ---- each input channel needs one token. And that's the default firing rule if no rate parameters are declared. It is also appropriate to mention that all SDF actors can be directly used in DDF domain since SDF domain is a subset of DDF domain.

This code block from the DDFSelect actor shows using ArrayToken to represent rates:

```
public boolean postfire() throws IllegalActionException {
        if (_isControlRead) {
            Token[] rates = new IntToken[input.getWidth()];
            for (int i = 0; i < input.getWidth(); i++) {
                rates[i] = new IntToken(0);
            }
            rates[_control] = new IntToken(1);
            input_tokenConsumptionRate.setToken(new ArrayToken(rates));
            control_tokenConsumptionRate.setToken(new IntToken(0));
        } else {
            Token[] rates = new IntToken[input.getWidth()];
            for (int i = 0; i < input.getWidth(); i++) {
                rates[i] = new IntToken(0);
            }
            input_tokenConsumptionRate.setToken(new ArrayToken(rates));
            control_tokenConsumptionRate.setToken(new IntToken(1));
        }
        return super.postfire();
}
```

Finally we give an example of non-sequential firing rules. The example is the famous Gustave function . It has three input ports and has the following firing rules:

$$\{([1],\ [0],\ \perp),([0],\ \perp,\ [1]),(\perp,\ [1],\ [0])\}$$

A blocking read on any input port would not give the desired behavior. An actor with this set of firing rules can not be used in the current DDF implementation.

3.4 Performance Comparison: SDF, DDF and PN

Dataflow process networks have been shown to be a special case of Kahn process networks. In dataflow process networks, each process consists of repeated firings of a dataflow actor, which defines a quantum of computation. Using this quantum avoids the complexities and context switching overhead of process suspension and resumption incurred in most implementations of Kahn process networks. Instead of context switching, dataflow process networks are executed by scheduling the actor firings. This scheduling can be done at compile time or at run time. SDF is scheduled at compile time. After the static schedule of an SDF graph is computed, the actors can be fired according to the predetermined sequence specified by the schedule. DDF is scheduled at run time, which is much more expensive.

To compare the performance of the SDF, DDF and PN domains, we simulated one SDF model under all three domains and plotted the running time in each domain. The model was slightly modified from ViterbiDecoder created by Rachel Zhou and Edward Lee so that we can better control the number of iterations in each domain. We also replaced original MonitorValue actors with Discard actors so that their graphical displays would not slow down the simulation. The final model contains 30 actors. As we can see from Figure 3.6, the simulation in the PN domain takes about twice as long as that in the DDF domain, and both are proportional to the number of iterations. SDF has an upfront cost

for computing static schedule. After that, the actual running time increases very slowly with the number of iterations. The simulation is done on a 2.4GHz Pentium 4 desktop installed with Windows XP Professional.

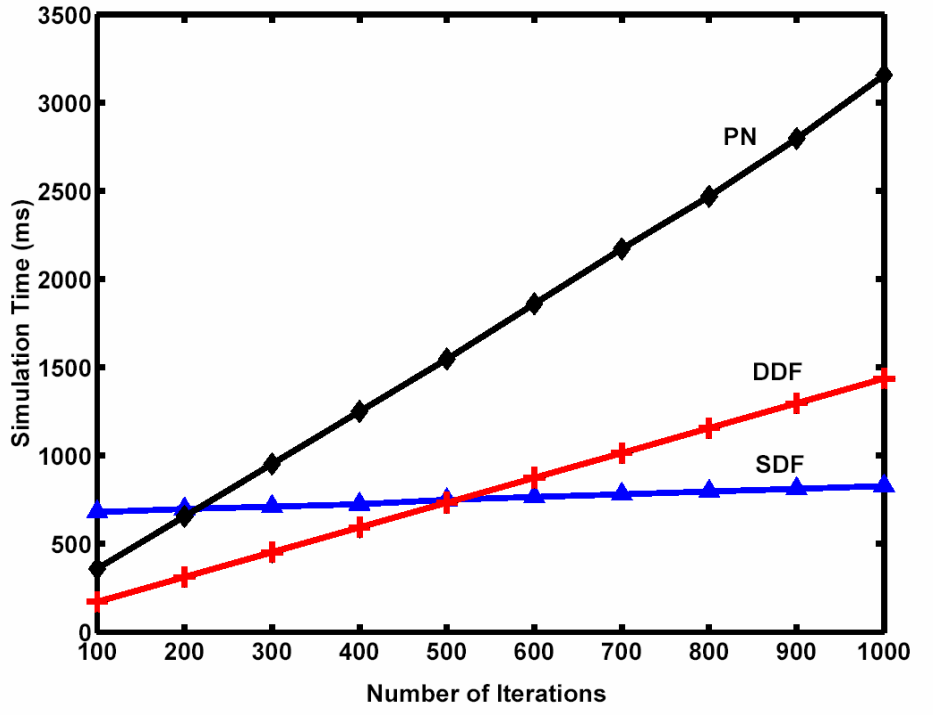


Figure 3.6 Simulation time of an SDF model under SDF, DDF, PN domains

3.5 Mixing DDF with Other Ptolemy II domains

The Ptolemy project advocates a system modeling and design paradigm called hierarchical heterogeneity [19][20]. There exist different approaches to solve the complexity of modern embedded system, which consists of heterogeneous components interacting with very different styles. One is the unified approach, which seeks a consistent semantics for the specification of the complete system. The other is the heterogeneous approach, which seeks to systematically combine disjoint semantics each describing the characteristics of a subsystem. Different semantics correspond to different models of computation (MoC). Each MoC usually matches well with certain design style

or certain aspects of the system. Small, specialized languages and tools based on different MoCs have proved to be very useful due to their appropriateness for certain domains of application and their formal properties resulting from the constrained semantics. Using hierarchy, one can divide a complex model into a tree of nested submodels with each level being homogeneous (each described by a particular MoC), thus allowing different MoCs used at different levels. Figure 3.7, which is directly borrowed from [21], shows a hierarchical model in Ptolemy II. Atomic actors, such as A1 and B1, appear at the bottom of the hierarchy. Composite actors, such as A2, can further contain other actors, so the hierarchy can be arbitrarily nested. Director2 is called the local director of A2 while Director1 is called the executive director of A2. Hierarchical heterogeneity is achieved by having different directors at different levels of the model.

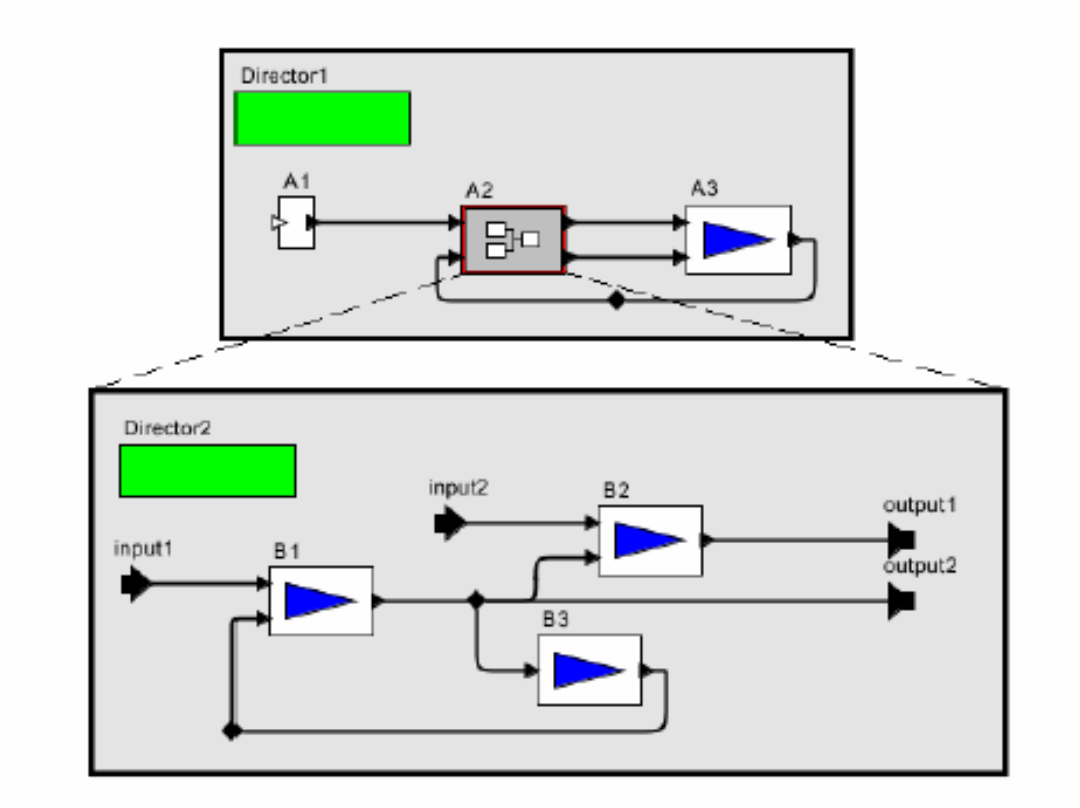


Figure 3.7 A hierarchical model in Ptolemy II

A key concept to achieve compositional execution is the notion of iteration for a model of computation. Using the terminology Jie Liu developed [22], an iteration is one composite precise reaction that starts with the trigger of the composite actor and finishes at a compositional quiescent state where each actor contained reaches its own quiescent state. Generally, how many individual precise reactions to aggregate into one composite precise reaction could be domain dependent. For some domains one iteration is natural and well-defined. For others it is not so obvious, which is unfortunately the case for DDF domain.

A composite actor can be embedded inside the DDF domain as long as the actor declares the consumption rate for each input port (or use the default value 1) and has a well-defined iteration when enabled and invoked. For example, an SDF composite actor contains an SDFDirector as its local director which (by delegating to the SDFScheduler) is responsible for statically analyzing the model contained by this composite actor and coming up with a complete cycle of component actor firings (if it exists), which would bring the composite actor to its original state (where all internal queues have the same number of tokens). Therefore a complete cycle for the SDF composite actor can be defined as one iteration and can be repeated indefinitely in bounded memory. Sometimes even when there is only one flat DDF model without any hierarchy, it is useful to compose some SDF actors into a submodel and add one level of hierarchy so that the submodel can be scheduled with static analysis. This would relieve some of the burden for dynamic scheduling. One should take caution in doing so, because arbitrary composition may lead to a deadlock which does not exist in the original model. Buck further developed this idea and created a new Boolean dataflow (BDF) domain [15]. This domain allows some special dynamic actors such as BooleanSelect and BooleaSwitch.

Automatic clustering can be performed to derive a quasi-static schedule for part of the model or even for the whole model if the clustering algorithm succeeds in reducing the model into a composite actor. If more than one actor remains to be scheduled, a DDFDirector can be used at the top level to direct the execution of the whole model, while each composite actor uses its own quasi-static schedule. This effectively reduces dynamic scheduling decisions that must be made during execution, because although quasi-static schedules still involve some data-dependent decisions, they come in a much simplified and disciplined form.

One the other hand, composition of actors will decrease the granularity of the model and may lead to sub-optimal performance when the actors are scheduled on, say, parallel processors. Therefore the overall effect must be analyzed to make a good decision.

When it comes to embedding DDF domain inside other domains, a lot of subtleties arise that fundamentally result from the fact that DDF domain is Turing-complete and a lot of properties are undecidable. As usual we need to define a meaningful iteration for the DDF submodel and also define firing rules for the composite DDF actor.

For a top level DDF model, we've already defined "basic iteration" and "iteration". If we use the basic iteration as one iteration for the DDF submodel, we can guarantee the reactivity of the composite actor since a basic iteration is always finite. However, we cannot guarantee that each iteration will produce fixed number of tokens, therefore a DDF submodel cannot be embedded in an SDF model. There is another problem with using the basic iteration. In Figure 3.8, Actor A would produce 2 tokens and actor B

would consume 1 token in each basic iteration (except the first iteration when B does not have token on its input port). This would lead to unbounded memory consumption. One solution is to define an iteration such that actor B is fired twice in each iteration (as done in a top level DDF model using parameter *requiredFiringsPerIteration*). But imagine the case that actor A produces a varying number of tokens in each firing. In this case it would be nice to define an iteration which consists of firing actors in the submodel until the submodel is deadlocked without reading tokens from outside domain. This option is indeed provided, as we will explain later. (The reader might argue the submodel in Figure 3.8 is actually an SDF composite actor and using DDFDirector is neither necessary nor desired in this case. But we could easily find a DDF composite actor with the same problem.)

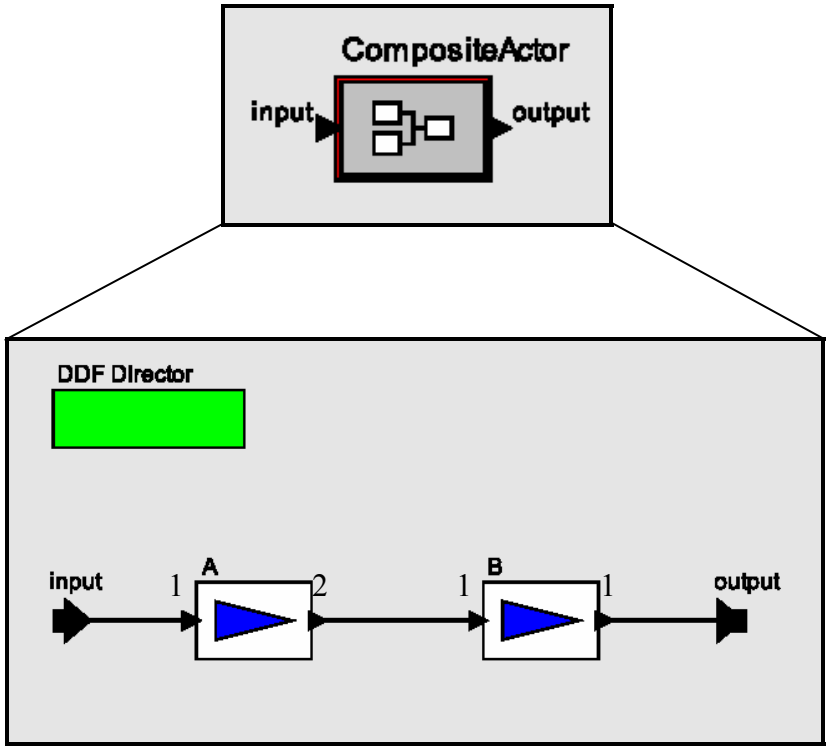
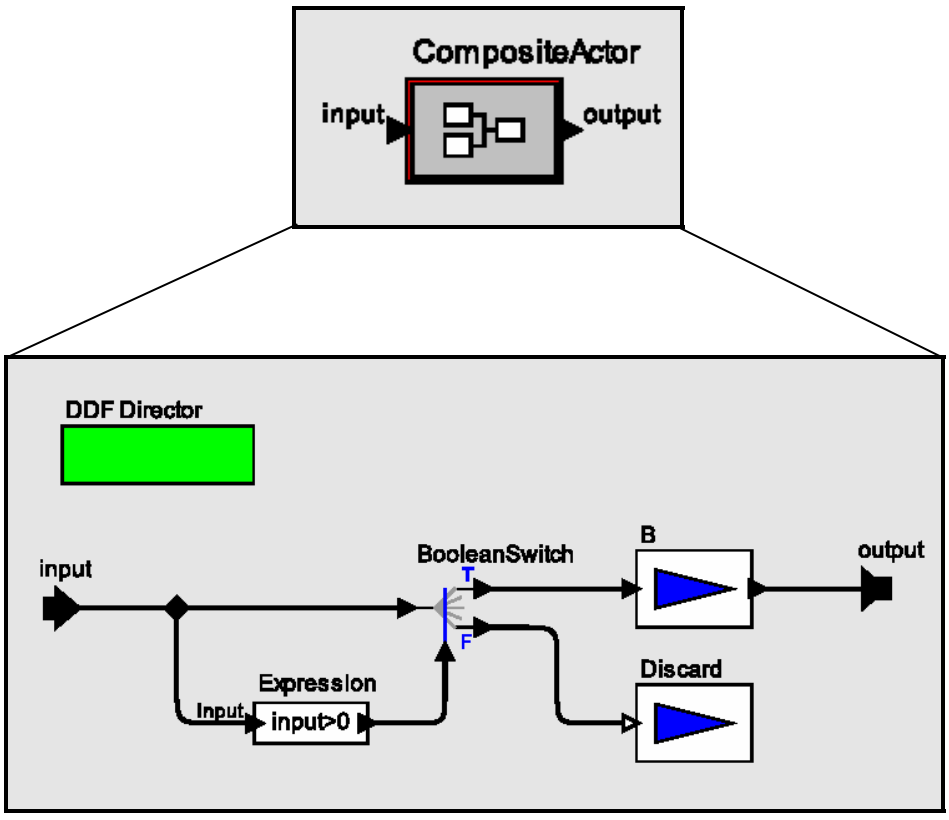


Figure 3.8 An example of a DDF composite actor

If we define an iteration for a DDF submodel such that some contained actor(s) must be fired a given number of times as we just did for Figure 3.8, it will have difficulty in other models. For example, in Figure 3.9, it would be nice to have actor B fire once in each iteration so that there is one output token per iteration. The problem is that we cannot define firing rules for such an iteration because we don't know how many tokens the composite actor needs to consume before it can output one token. Again in this case one iteration can be defined as running the submodel until deadlock. Then it has a well-defined firing rule, i.e., the input port needs one token. But it cannot produce a token per iteration. That's a compromise we have to make because the DDF domain is a superset of SDF domain and it's generally impossible to convert a DDF submodel into an SDF composite actor.



3.9 Yet another example of a DDF composite actor

So far it seems that running until deadlock is a preferred choice for defining an iteration for a DDF submodel. However, in the previous two examples, we did not include any source actor nor feedback loop, which can destroy reactivity of the composite actor if we don't constrain the number of times the actors in the submodel can fire. In Figure 3.10, a nice way to define one iteration is to add *requiredFiringsPerIteration* parameter to DDFBooleanSelect actor with value two. (Remember two-phase firings of DDFBooleanSelect actor which outputs one token every two firings.)

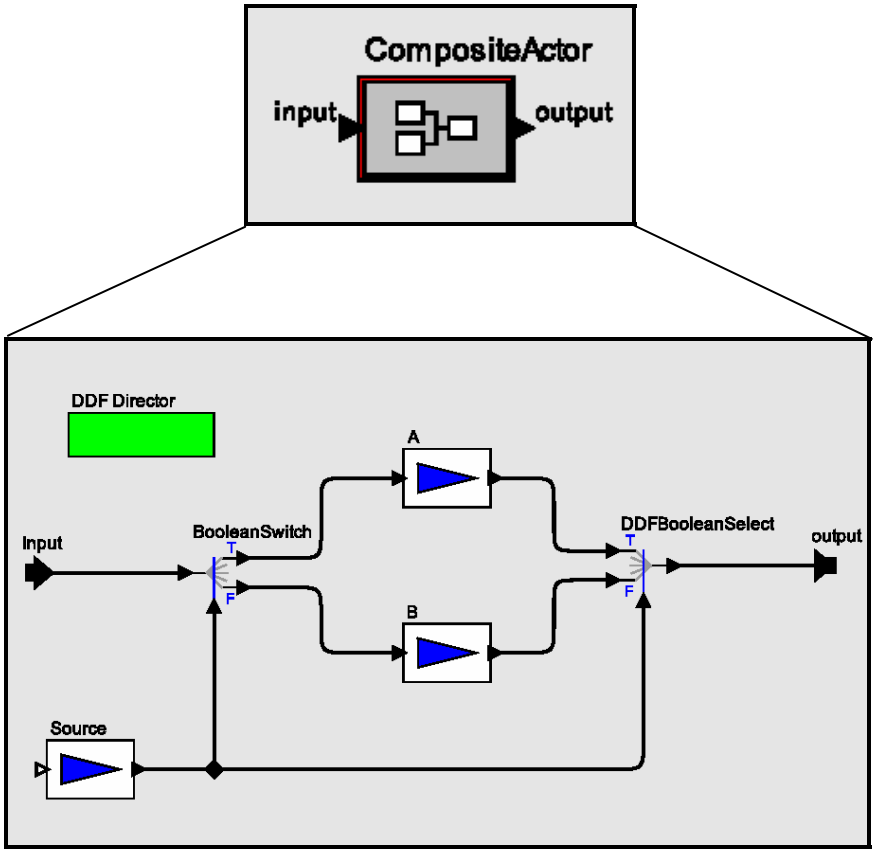


Figure 3.10 Still another example of a DDF composite actor

Having seen the complexity of defining one iteration for a DDF composite actor, we choose to provide a number of choices to the user, and let the user take the final responsibility to define one iteration, as appropriate for their specific application. To sum up, an iteration can be defined as (1) a basic iteration, (2) several basic iterations which

satisfy *requiredFiringsPerIteration* for the actors with that parameter, (3) repeating a basic iteration until the submodel is deadlocked. Among these three definitions, (2) overrides (1), (3) overrides (1) and (2). Also notice that while definition (1) and (2) are also applicable for top level DDF models, definition (3) is only applicable for embedded DDF submodels.

We also have the following rules for transferring tokens into and out of DDF submodels. If the user manually adds *tokenConsumptionRate* and *tokenProductionRate* parameters to the input ports and output ports of the DDF composite actor, the local DDFDirector will leave them as they are and obey them assuming the user knows the consequence of his(her) actions. If the rates of some ports are not manually defined, the local DDFDirector will try to define them in a reasonable way. For the input port, the *tokenConsumptionRate* is set to the minimum number of tokens needed to satisfy one of destination actors connected to this input port. Therefore the rate could be zero if one of destination actors already has enough tokens on the channel connected to this input port. However, if no actor in the DDF composite actor gets fired during the previous iteration, the *tokenConsumptionRate* is set to the minimum number of tokens needed to satisfy one of destination actors that are connected to this input port and haven't been satisfied on the connecting channels. The motivation behind this seemingly complicated procedure is to try to break a deadlock using minimum number of tokens. When every destination actor connected to this input port has been satisfied and the submodel is still deadlocked in the previous iteration, the *tokenConsumptionRate* of this input port is set to zero. For the output port (which has no *tokenProductionRate* parameter added by the user), the local DDFDirector will not set the rate for this output port because in general there is no way

to know how many tokens will be produced before each iteration. Instead the local DDFDirector will transfer all tokens produced for this output port in each iteration.

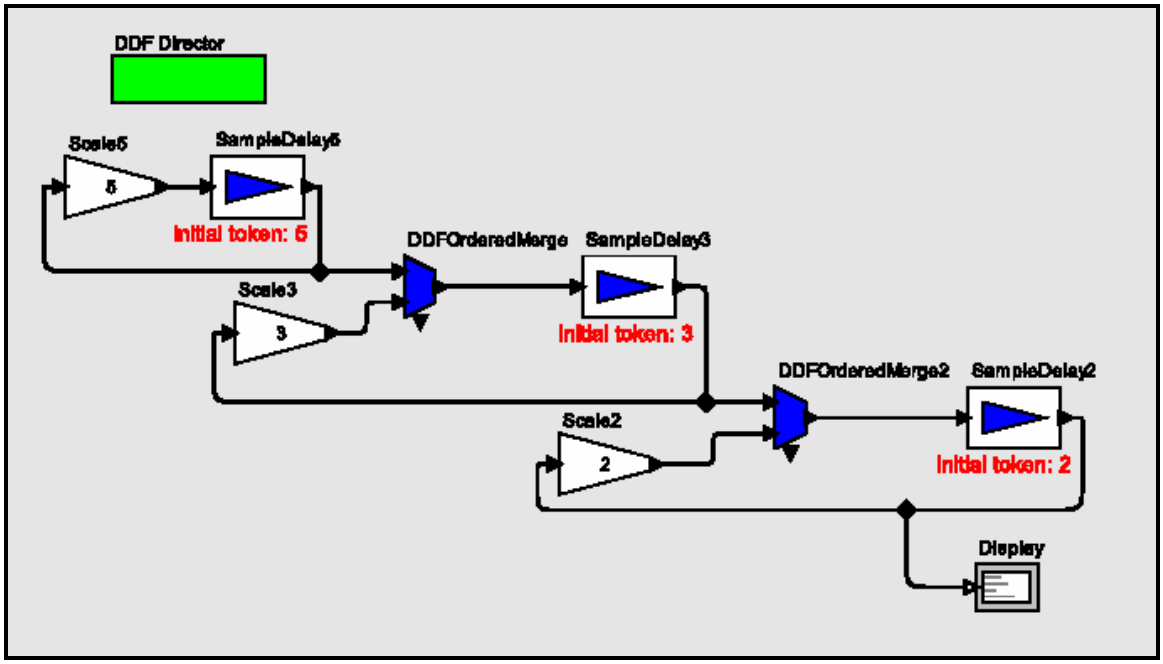
Finally we want to emphasize that the mechanism we provide to define iterations and set rates should be considered “best effort”. We don’t claim this will solve the problem of embedding arbitrary DDF submodel into other domains because fundamentally DDF domain is Turing-complete and many properties are undecidable. However, many interesting models can be built if the mechanism is used wisely. We will show some examples in the next chapter.

4 Application Examples

This section gives several examples to show how conditionals, data-dependent iterations, recursion and other dynamic constructs can be modeled in DDF domain.

4.1 An OrderedMerge example

The first example in Figure 4.1 is due to Kahn and MacQueen. It calculates integers whose prime factors are only 2, 3 and 5, with no redundancies. It uses the OrderedMerge actor, which takes two monotonically increasing input sequences and merges them into one monotonically increasing output sequence. This model was originally implemented in PN domain and is adapted here to show the concept of an iteration in DDF domain. Although no static or quasi-static schedule can be computed for this model, by adding a *requiredFiringsPerIteration* parameter to Display actor and setting its value to one, we have effectively defined one iteration of this model such that there is one output per iteration. This iteration does not bring the model to its original state or anything close to that, but matches well with our intuition and serves to precisely control the progress of the simulation by specifying the number of iterations (i.,e. how many output tokens) we want in one execution.



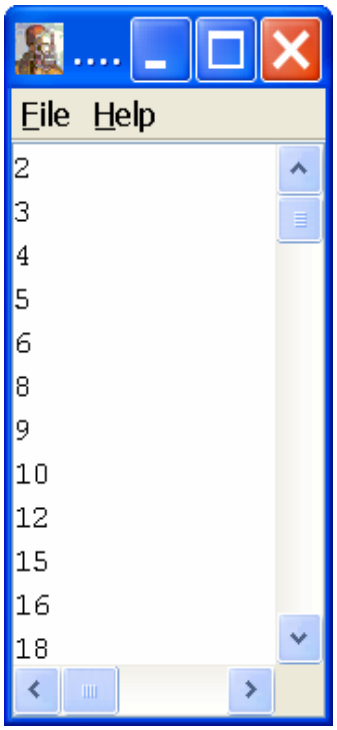
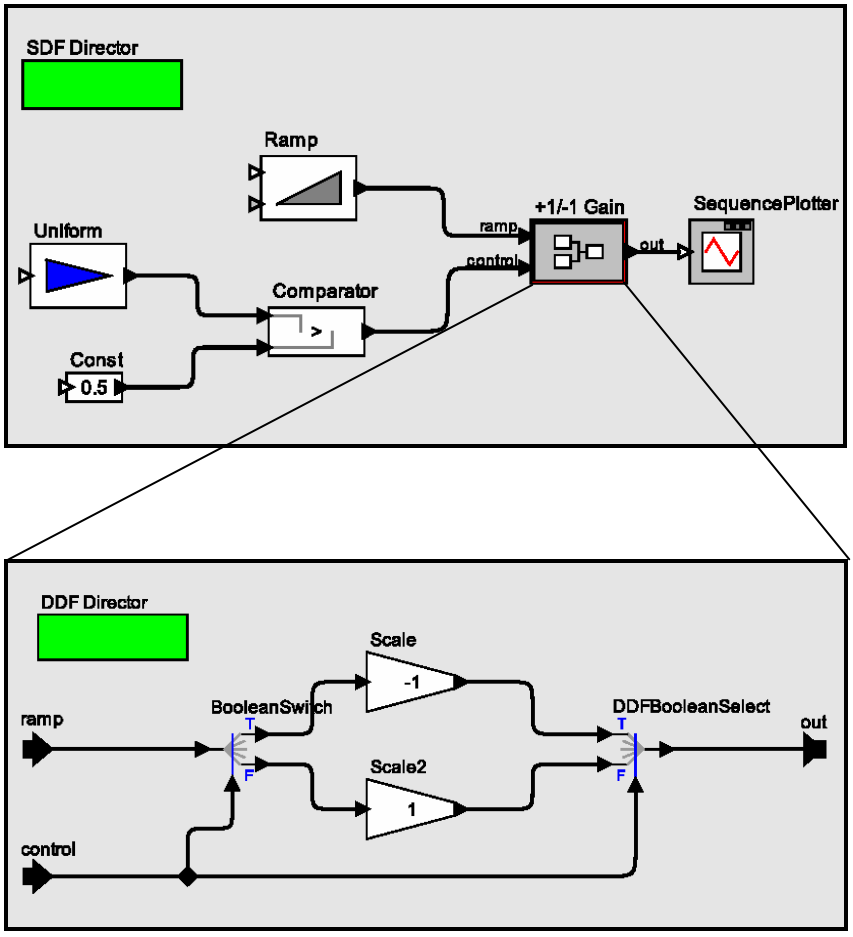


Figure 4.1 An OrderedMerge example

4.2 Conditionals with If-Else Structure

This DDF-inside-SDF example in Figure 4.2 demonstrates an if-then-else like structure in the dataflow context.



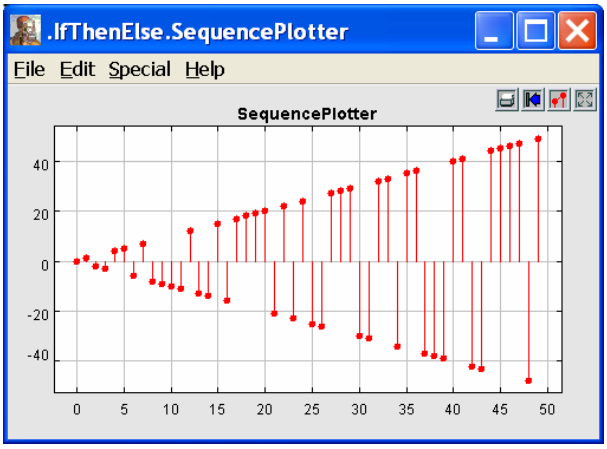


Figure 4.2 An example with if-then-else structure

In this example, we must add rate parameters and set their values to one for all input ports and output port of the DDF composite actor because it must look like an SDF actor from outside. We also add a *requiredFiringsPerIteration* parameter to DDFBooleanSelect actor and set its value to 2. In each iteration of the DDF composite actor, each input port transfers a token to the inside. Depending on the Boolean value of the token transferred from *control* input port, the token transferred from *ramp* input port is routed to the upper or the lower branch. Then it is processed by some actor(s) and reemerges from DDFBooleanSelect actor. Finally it is transferred to the outside through the output port.

4.3 Data-Dependent Iterations

The model in Figure 4.3 illustrates a do-while like structure in the dataflow context. Each input integer from Ramp actor is repeatedly multiplied by 0.5 until the product is below 0.5. The outside-the-loop Plotter has a *requiredFiringsPerIteration* parameter with value 1. Therefore each iteration in this model corresponds to a complete do-while loop for each input integer.

Another model in Figure 4.4 generates a random walk with evenly spaced steps. It has three-level nesting: SDF inside DDF inside SDF. In each iteration of DDF composite actor, the ParameterPort *numberOfSteps* gets one token from outside domain and uses its value to configure both Repeat actors. Since the number of tokens produced by the Repeat actors varies, we set the *runUntilDeadlockInOneIteration* parameter of DDFDirector to true so that in each iteration of DDF composite actor, all tokens produced by Repeat actors are consumed.

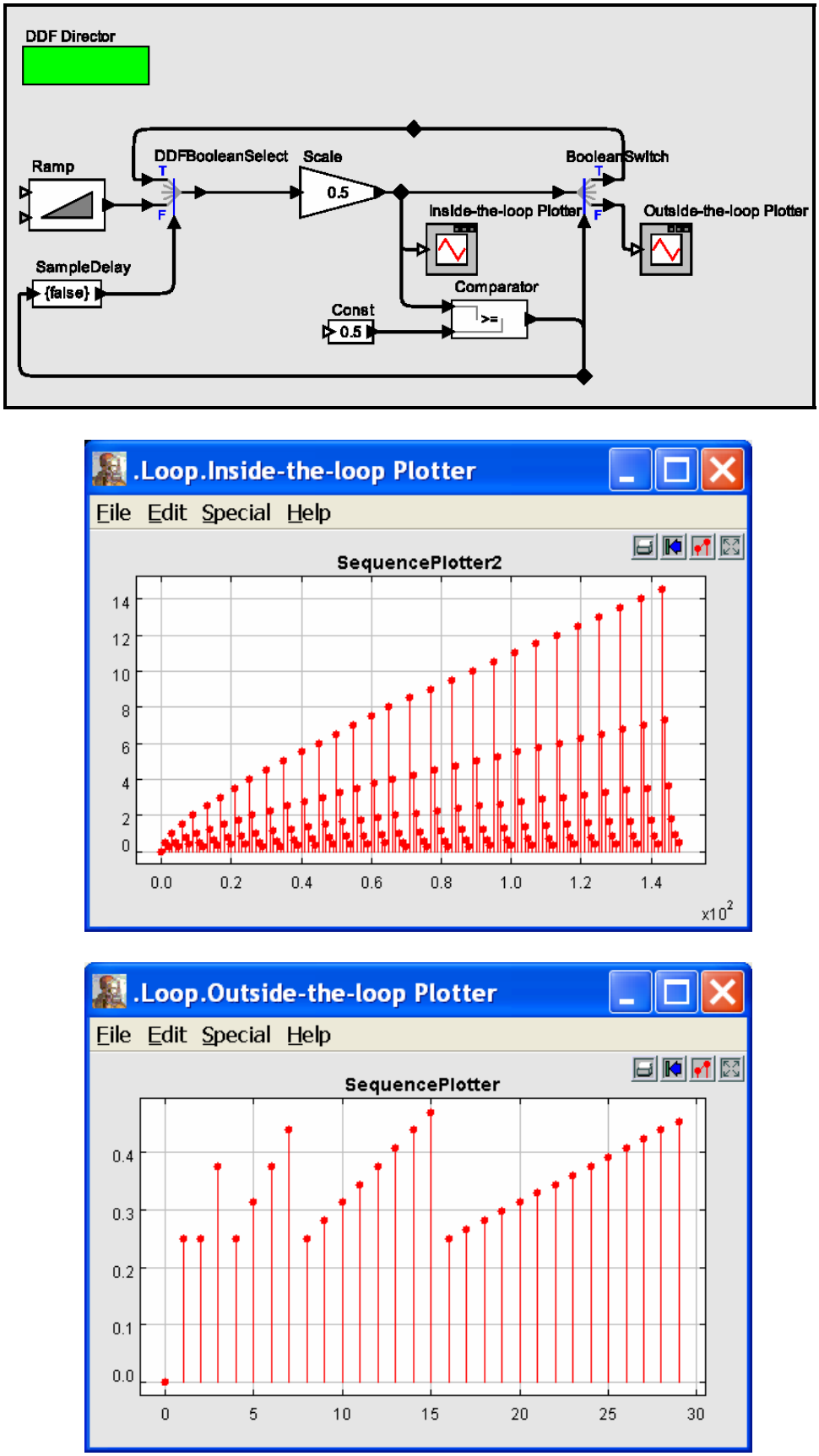


Figure 4.3 An example with do-while structure

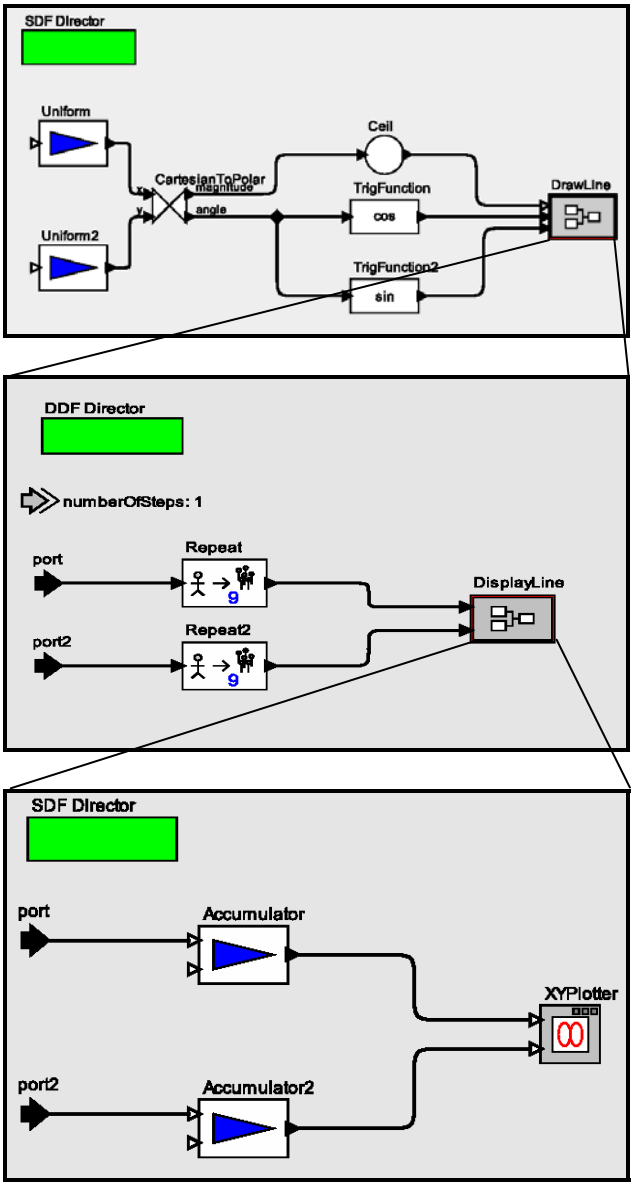


Figure 4.4 A random walk example with SDF/DDF/SDF nesting

4.4 Recursion: Sieve of Eratosthenes

Eratosthenes (276 - 196 B.C.) invented a method called Sieve of Eratosthenes for efficiently constructing tables of prime numbers, which, in modified form, is still an important tool in number theory research. It is interesting to see it implemented in a dataflow context. The method goes like this:

First, write down a list of integers beginning with 2,

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Then filter out all multiples of 2:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
    x   x   x   x    x    x    x    x    x    x
```

Move to the next remaining number, which in this case is 3, then filter out all its multiples:

```
2 3 4 5 6 7 8 9  10 11 12 13 14 15 16 17 18 19 20
    x   x   x x  x    x    x  x  x    x    x
```

Continue in this fashion and filter out all multiples of the next remaining number. The numbers that are left up to this remaining number are all prime numbers. In principle, this process can be repeated indefinitely to find any prime number.

This method can be implemented in a dataflow model with an ActorRecursion actor we created. Because each filtering has the same structure, we only need to build one structure and whenever filtering against another number is needed, we can copy (clone) the same structure and expand the topology.

Technically, the ActorRecursion actor is a composite actor with a StringParameter named *recursionActor*. Upon firing, it clones the composite actor containing itself and referred

to by its StringParameter. It then places the clone inside itself and connects the corresponding ports of both actors. It uses local DDFDirector to preinitialize the clone and then transfers all tokens contained by input ports of this actor to the connected opaque ports inside. It again uses local DDFDirector to initialize all actors contained by this actor and check their statuses (enabled? deferable?). It then transfers all tokens (produced during initialization) contained by output ports of this actor to the connected opaque ports outside. It finally merges local DDFDirector with its executive DDFDirector and then removes local DDFDirector. Thus during entire execution this actor is fired at most once, after which the executive director directly controls all actors inside.

As is done in many cases, the Display actor in the model (see Figure 4.5) has a *requiredFiringsPerIteration* parameter and the value is one. Therefore in each iteration it will get a prime number. Before the execution, ActorRecursion contains only an embedded DDFDirector. But as execution goes on, the model topology will be dynamically expanded.

We have built another demo called Hanoi Tower using the same ActorRecursion. Due to its complexity, it is not shown in this report. Interested readers may refer to Ptolemy tree and future distribution.

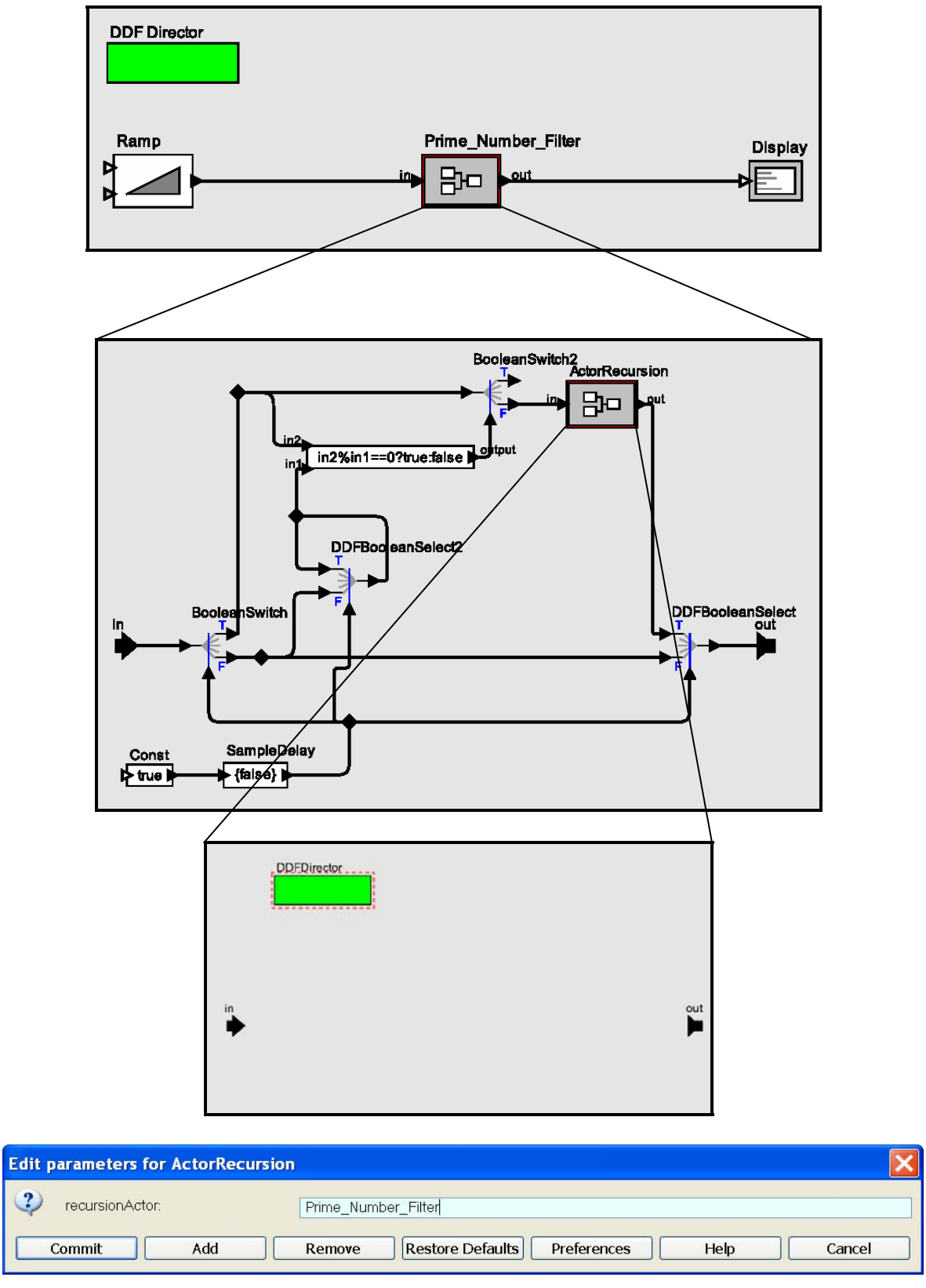


Figure 4.5 Implementation of Sieve of Eratosthenes with recursion

5 Conclusion and Future Work

In this report, we first motivated the use of dataflow model of computation and gave a brief review of the denotational semantics of Kahn process networks and a denotational semantics of dataflow with firing as theoretical foundations. Then we described the implementation of a dynamic dataflow scheduling algorithm under Ptolemy II framework guided by several criteria we'd like it to satisfy. We show how to write actors that can be used in this domain. We also discuss composing DDF with other models of computation. The report concludes with several examples showing how conditionals, data-dependent iterations, recursion and other dynamic constructs can be modeled in DDF domain.

The future work includes developing a scheduling algorithm for non-sequential firing functions such as Gustave function. Mechanisms for composing the DDF domain with other models of computation could be further improved. A timed DDF model of computation similar to timed PN could be developed, thus making it easier to compose with other timed models of computation.

References

[1] Edward A. Lee, "Embedded Software," to appear in Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

[2] G. Berry, "Real Time programming: Special purpose or general purpose languages," in Information Processing, Ed. G. Ritter, Elsevier Science Publishers B.V. (North Holland), vol. 89, pp. 11-17, 1989.

[3] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," IEEE Transactions on CAD, Vol. 17, No. 12, December 1998.

[4] http://ptolemy.eecs.berkeley.edu/

[5] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (eds.), "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)," Technical Memorandum UCB/ERL M04/27, University of California, Berkeley, CA USA 94720, July 29, 2004.

[6] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (eds.), "Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture)," Technical Memorandum UCB/ERL M04/16, University of California, Berkeley, CA USA 94720, June 24, 2004.

[7] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (eds.), "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)," Technical Memorandum UCB/ERL M04/17, University of California, Berkeley, CA USA 94720, June 24, 2004.

[8] E. A. Lee, "A Denotational Semantics for Dataflow with Firing," Memorandum UCB/ERL M97/3, Electronics Research Laboratory, U. C. Berkeley, January 1997.

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

[10] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.

[11] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," Proc. of the IEEE, September, 1987.

[12] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete, "Cyclo-static data flow: Model and implementation," Proc. 28th Asilomar Conf. on Signals, Systems, and Computers, pp. 503-507, 1994.

[13] 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.

[14] Bhattacharya, B. and S. S. Bhattacharyya, "Parameterized Dataflow Modeling of DSP Systems," International Conference on Acoustics, Speech, and Signal Processing. Istanbul, Turkey, June 2000.

[15] J. T. Buck, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model," Technical Memorandum UCB/ERL 93/69, Ph.D. Thesis, Dept. of EECS, University of California, Berkeley, CA 94720, 1993.

[16] T. M. Parks, "Bounded Scheduling of Process Networks," Technical Report UCB/ERL-95-105, PhD Dissertation, EECS Department, University of California. Berkeley, CA 94720, December 1995.

[17] E.A. Lee, S. Bhattacharyya, J.T. Buck, W.T. Chang, M.J. Chen, B.L. Evans, E.E. Goei, S. Ha, P. Haskell, C.T. Huang, W.J. Huang, C. Hylands, A. Kalavade, A. Kamas, A. Lao, E.A. Lee, S. Lee, D.G. Messerschmitt, P. Murthy, T.M. Parks, J.L. Pino, J. Reekie, G. Sih, S. Sriram, M.P. Stewart, M.C. Williamson, K. White, "The Almagest," five volumes of documentation for Ptolemy Classic, a heterogeneous simulation and design environment supporting multiple models of computation and the predecessor to Ptolemy II, a Java-based environment.

[18] Marc Geilen, Twan Basten, "Requirements on the Execution of Kahn Process Networks," Proc. European Symposium on Programming Languages and Systems, pp. 319-334, April 7-11, 2003. Available in Lecture Notes in Computer Science 2618, Springer, Berlin, Germany, 2003.

[19] W.-T. Chang, S.-H. Ha, and E. A. Lee, "Heterogeneous Simulation -- Mixing Discrete-Event Models with Dataflow," invited paper, Journal on VLSI Signal Processing, Vol. 13, No. 1, January 1997.

[20] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, Yuhong Xiong, "Taming Heterogeneity---the Ptolemy Approach," Proceedings of the IEEE, v.91, No. 2, January 2003.

[21] Jie Liu, Johan Eker, Jörn W. Janneck, Xiaojun Liu, and Edward A. Lee, "Actor-Oriented Control System Design: A Responsible Framework Perspective," to appear in IEEE Transactions on Control System Technology, Draft version, March, 2003.

[22] Jie Liu, "Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems," Ph.D. Thesis, Technical Memorandum UCB/ERL M01/41, University of California, Berkeley, CA 94720, December 20th, 2001.