UCB/ERL Memorandum M93/69, September 1993

SCHEDULING DYNAMIC DATAFLOW GRAPHS WITH BOUNDED MEMORY USING THE TOKEN FLOW MODEL

by

Joseph Tobin Buck

B. E. E. (Catholic University of America) 1978
M.S. (George Washington University) 1981

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:
Prof. Edward A. Lee, chair
Professor David A. Messerschmitt
Professor Sheldon M. Ross

1993

Abstract

SCHEDULING DYNAMIC DATAFLOW GRAPHS WITH BOUNDED MEMORY USING THE TOKEN FLOW MODEL

by

Joseph Tobin Buck

Doctor of Philosophy in Electrical Engineering

Prof. Edward A. Lee, chair

This thesis presents an analytical model of the behavior of dataflow graphs with data-dependent control flow. In this model, the number of tokens produced or consumed by each actor is given as a symbolic function of the Boolean-valued tokens in the system. Several definitions of consistency are discussed and compared. Necessary and sufficient conditions for bounded-length schedules, as well as sufficient conditions for determining whether a dataflow graph can be scheduled in bounded memory are given. These are obtained by analyzing the properties of minimal cyclic schedules, defined as minimal sequences of actor executions that return the dataflow graph to its original state. Additional analysis techniques, including a clustering algorithm that reduces graphs to standard control structures (such as "if-then-else" and "do-while") and a state enumeration procedure, are also described. Relationships between these techniques and those used in Petri net analysis, as well as in the theory of certain stream languages, are discussed.

Finally, an implementation of these techniques using Ptolemy, an object-oriented simulation and software prototyping platform, is described. Given a dynamic dataflow graph, the implementation is capable either of simulating the execution of the graph, or generating efficient code for it (in an assembly language or higher level language).

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I wish to acknowledge and thank Professor Edward Lee, my thesis advisor, for his support, his leadership, and his friendship, and for the ideas that helped to inspire this work. I thank Professor David Messerschmitt for serving as a second advisor to me, and thank both Lee and Messerschmitt for conceiving of the Ptolemy project and giving me the opportunity to play a key role. I also thank Professor Sheldon Ross for serving on my committee.

I thank my colleagues Tom Parks and Shuvra Bhattacharrya for their careful review of earlier drafts of this dissertation and their useful suggestions. I benefited greatly by working closely with Soonhoi Ha, my collaborator on many projects and papers. S. Sriram assisted in clarifying several points relating to computability theory. I also benefitted from technical interaction with my colleagues Wan-teh Chang, Paul Haskell, Philip Lapsley, Asawaree Kalavade, Alan Kamas, Praveen Murthy, José Pino, and Kennard White, as well as the feedback from all those brave enough to use the Ptolemy system.

This work was supported by a grant from the Semiconductor Research Corporation (93-DC-008).

I cannot conceive of how I could have accomplished what I have without the support and love of my wife, Christine Welsh-Buck. I dedicate this work to her.

1

THE DATAFLOW PARADIGM

I believe that the current state of the art of computer programming reflects inadequacies in our stock of paradigms, in our knowledge of existing paradigms, and in the way our programming languages support, or fail to support, the paradigms of their user communities.

— R. Floyd

This dissertation is devoted to the application of a particular model of computation, namely dataflow, to the solution of problems in digital signal processing (DSP). It is not our intent to dogmatically insist that any particular model be applied in a pure form; rather, it is our thesis that the most efficient applications of dataflow to DSP use a hybrid model, combining the best features of dataflow and other models of computation, and that it is advantageous to determine as much as possible about the execution of a dataflow system at "compile time". Therefore this section is an attempt to place the dataflow paradigm in context with respect to other possibilities and to flesh out the theoretical background for the graphical and stream-based models of computation we will consider.

In section 1.1, we discuss the distinction between operational and definitional paradigms in computer science, building a case for consideration of definitional approaches to problem formulation in computer science. A variety of operational and definitional models are discussed. In section 1.2, we focus on those definitional models that can be expressed graphically, most of which are related in some way to the Petri net model. These models, for the most part, form the basis of dataflow computing. The rest of the chapter presents a survey of dataflow computing from both the hardware and software perspectives: section 1.3 discusses dataflow machines, and section 1.4 discusses languages that implement a dataflow model. Finally, section 1.5 summarizes the chapter.

Following Floyd [Flo79], we adopt the term *paradigm* from Thomas Kuhn's *The Structure of Scientific Revolutions*. A Kuhnian paradigm, in the field of history of science, is a work that shares two characteristics: it succeeds in attracting an enduring group of adherents away from competing modes of scientific activity, and it is sufficiently open-ended to leave all sorts of problems for the "converts" to solve [Kuh62]. By analogy, in computer science we can say that structured programming is a paradigm (Floyd's main example), as is object-oriented programming, logic programming, communicating sequential processes, and many others. Floyd also identifies techniques with more limited applicability as paradigms, thus branch and bound or call by name are paradigms.

Ambler *et al*. identify three levels of programming paradigms [Amb92]: those that support high-level approaches to design (functional languages, object-oriented design), methods of algorithm design, and low-level techniques (copying versus sharing of data, for example). We are mainly concerned with high-level paradigms, but unlike Ambler, we will consider both programming language paradigms and those that pertain to computer architecture. In general, we have a hierarchy of languages: at the highest level, the user or system designer manipulates the most abstract objects. Any number of intermediate levels may intervene between this model and the physical machine, and paradigms model the organization and design of each level.

1.1. OPERATIONAL VS DEFINITIONAL

Whether we consider programming languages or computer architecture and organization, it appears that one distinction is fundamental: the difference between operational and definitional approaches to problem-solving. Roughly stated, the distinction has to do with the level of detail in which the designer or programmer must specify how the answer is computed, in addition to specifying what is computed. This distinction is similar to, but not the same as, the distinction between imperative and declarative models of programming made by the advocates of functional programming (for example, [Hud89]).

1.1.1 The Operational Paradigm

The most successful paradigm for computer architecture and organization is the von Neumann model of the computer. The most important aspect of this model for our purposes is that the von Neumann machine has a state, corresponding to the contents of memory and of certain internal registers in the processor (the program counter, for example). The machine executes one instruction at a time in a specified order, and the result of each instruction is that one or more memory locations and internal registers take on a new value.

The most commonly used computer languages have retained this fundamental paradigm: the programmer is presented with a higher-level and cleaner version of a von Neumann machine, and the task of the programmer is to specify the states and to schedule the state transitions. Following Ambler *et al.*, we refer to programming paradigms in which the designer or programmer specifies the flow of control that converts the starting state into the solution state by means of a series of state transitions as *operational*.

Given this definition, there are a great variety of programming languages and paradigms that fall under the operational approach, from unstructured assembly language to structured programming to object-oriented programming. Ambler *et al* divide traditional operational programming languages into two principal groups: imperative and object-oriented. Languages that support abstract types and information hiding but not inheritance, such as Ada, would fall in the latter group according to their classification, although other authors, notably Booch in [Boo91], call such languages *object-based*. The difference between imperative and object-based languages is mainly that the states have become much more abstract in object-based languages.

Parallel languages in which the programmer explicitly controls the threading to some degree are also considered operational. We will not discuss such languages further; the interested reader is directed to [Bal89].

While operational, imperative languages are very widely used, and many software engineering techniques have been developed to make them more manageable, there are some significant disadvantages. As pointed out by Backus [Bac78], the imperative state transition model renders programming as well as programming execution intractable to formal reasoning. To be fair, there are techniques for reasoning about sequential programs provided that some structure is followed, as Dijkstra, Floyd, Hoare and others have shown. There are also languages that are explicitly based on a state machine model, such as Esterel [Ber92] and Statecharts [Har87], but they represent definitional (or pseudo-definitional) rather than operational approaches, since the programmer uses the language to specify properties the solution is to have and does not specify the exact sequence of steps in finding the solution. From an organizational point of view, programs for a state transition machine constitute rather sophisticated work schedules [Klu92], and efforts to reason about programs must deal with the fact that the specification of the exact order in which operations are to be performed can get in the way of the logic.

Despite these disadvantages, the very aspects that cause difficulties for the imperative specification of large parallel systems (the need to precisely specify all details, together with their order) often turn into advantages when it is necessary to obtain the maximum performance for a particular small piece of code on a particular piece of hardware. As we will later see, certain hybrid models (e.g. coarse-grain dataflow as in block diagram languages and the cooperating sequential processes model of [Kah74]) may be used to combine aspects of the operational and definitional approaches.

1.1.2 Definitional and Pseudo-definitional Models

In the definitional or declarative paradigm, we express the result we wish to produce by defining it rather than by giving a step-by-step method of computing it. Relationships between inputs and the required output are specified in a formal manner, and inputs are transformed to outputs by state-independent means. In principle, the programmer does not specify the order of operations, but in many cases mechanisms are provided to "cheat" and hence we use the term *pseudo-definitional* to describe the hybrid approach that results.

The canonical example of this paradigm is one of the oldest, that subset of Lisp known as "pure Lisp". In this subset, results are computed as a result of function application alone; there is no assignment (other than the binding of formal arguments to actual parameters), no side effects, and no destructive modification of list storage. Results are generated by copying, and garbage collection is used to reclaim memory without intervention by the programmer. This is a simple example of functional programming, where the key concept is that of functional composition, feeding the result of one function to the next until the desired result is computed.

The major categories of definitional paradigms that we consider here include forms-based programming, logic programming, functional programming, and dataflow and stream approaches. Forms-based programming, as is used in spreadsheets, may well be the most common form of definitional programming in existence today, if we consider the sheer numbers of "programmers" (many of whom do not realize that they are in fact programming).

In the logic programming paradigm, we are given known facts, relationships, and rules of inference, and attempt to deduce particular results. Just as functions are the key to functional programming, relations are the key to logic programming. "Thus, logic programming from the programmer's perspective is a matter of correctly stating all necessary facts and rules [Amb92]." Evaluation of a logic program starts from a goal and attempts to deduce it by pattern matching from known facts or deduction from the given rules. In principle, this makes logic programming purely definitional, but because of the combinatorial explosion that results almost all logic programming languages have means of implicitly controlling the order of evaluation of rules, including mechanisms known as "cuts" to inhibit backtracking. Use of these mechanisms is essential in the logic programming language Prolog, for example [Mal87].

Functional programming languages are characterized by the lack of implicit state (state is carried around explicitly in function arguments), side effects, and explicit sequencing. Modern functional languages are characterized by higher-order functions (functions are permitted to return functions and accept functions as arguments), lazy evaluation (arguments are evaluated only when needed) as opposed to eager evaluation (in which arguments are always evaluated before passing them to functions), pattern matching, and various kinds of data abstraction [Hud89]. Functional languages possess the property known as *referential transparency*, or "equals may be replaced by equals"; this is a powerful tool for reasoning about and for transforming functional programs.

In the dataflow paradigm, streams of data flow through a network of computational nodes; each node accepts data values, commonly called *tokens*, from input arcs and produces data values on output arcs. The programmer specifies the function performed at each node. The only constraints on order of evaluation are those imposed by the data dependence implied by the arcs between nodes. Visual representations for this kind of computation are natural; in addition, there are textual representations for such languages that are typically translated into a dataflow graph internal form.

Dataflow languages are, for the most part, functional languages, distinguished mainly by their emphasis on data dependency as an organizing principle. Like functional languages, they are applicative, rather than imperative; many lack the notion of a higher-order function (a function that operates on and returns functions). In several dataflow languages, a distinguishing feature is the use of identifiers to represent conceptually infinite streams of data; this feature apparently originated in the language Lucid [Ash75]. The best-known languages of this type are Lucid, Val [Ack79] and its successor SISAL [McG83], and Id [Arv82] and its successor, Id Nouveau [Nik86]. We will explore the features of these languages and others in more detail in the next section.

Dataflow machines and graph reduction engines are examples of machines that implement definitional programming paradigms directly in hardware. We will have more to say about dataflow machines later in this thesis (see section 1.3); for a discussion of graph reduction engines see [Klu92].

1.2. GRAPHICAL MODELS OF COMPUTATION

Graphical models of computation are very effective in certain problem domains, particularly digital signal processing and digital communication, because the representation is natural to researchers and engineers. These models naturally correspond to dataflow semantics, resulting in many cases in definitional models that expose the parallelism of the algorithm and provide minimal constraints on the order of evaluation. Even where text-based rather than graphical languages are used (as in section 1.4), compilers often create graphical representations as an intermediate stage. Almost all graphical models of computation can be formulated as either special cases of, or in some cases, generalizations of Petri net models, including the dynamic dataflow models that are the core of this thesis. This section introduces the analysis techniques that provide tools for understanding and manipulating these models.

1.2.1 Petri Nets

Petri nets are a widely used tool for modelling, and several models of dataflow computation are important special cases of Petri nets. Before explaining the special cases, we will discuss Petri nets in their general form, using the definition of Peterson [Pet81].

A Petri net is a directed graph, $G = (V, A)$, where $V = \{v_1, \ldots, v_s\}$ is the set of vertices and $A = \{a_1, \ldots, a_r\}$ is a bag (not a set) of arcs.[1] The set V of vertices can be partitioned into two disjoint sets P and T, representing two different types of graph nodes, known as *places* and *transitions*. Furthermore, every arc in a Petri net either connects a place to a transition, or a transition to a place (no edge may connect two nodes of the same type). Thus if $a_i = (v_j, v_k)$ is an arc, either $v_j \in P$ and $v_k \in T$ or $v_j \in T$ and $v_k \in P$. There may be more than one arc connecting a given place to a given transition, or vice versa[2]; thus A is a bag rather than a set, and the membership function for a given node pair specifies the number of parallel arcs present between that pair of nodes.

In addition, places may contain some number of *tokens*. A *marking* of a Petri net is simply a sequence of nonnegative integers, one value per place in the net, representing the number of tokens contained in each place. It can be considered a function from the set of places P to the set of non-negative integers N, $\mu: P \to N$. A Petri net together with a marking is called a *marked Petri net*.

For each transition t in a Petri net, there is a corresponding set of input places $I(t)$ (the set of places for which an arc connects the place to the transition) and a set of output places $O(t)$ (the set of places for which an arc connects the transition to the place). Similarly, we can define the set of input transitions and output transitions for each place, $I(p)$ and $O(p)$.

The execution of a marked Petri net is controlled by the presence or absence of tokens. A Petri net executes by firing transitions. When a transition fires, one token is removed from each input place of the transition (if there are n parallel arcs from a place to a transition, then n tokens are removed from the place) and one token is added to each out-

1. A bag is distinguished from a set in that a given element can be included n times in a bag, so that the membership function is integer-valued rather than Boolean-valued. A discussion of bag theory as an extension of set theory as it applies to Petri nets appears in [Pet81].
2. In Petri's original formulation, parallel arcs were not permitted; we use the more general form discussed in Peterson [Pet81] and, following Peterson, use the term *ordinary Petri net* to discuss the more restricted case.

put place of the transition (again, if there are *n* parallel arcs from the transition to the same output place, *n* tokens are added to that place). The number of tokens in a given place can never be negative, so a transition may not fire if there are not enough tokens on any of its input places to fire the transition according to these rules. A transition that has enough tokens on all of its input places for it to fire is said to be enabled. Enabled transitions may fire, but are not required to; firings may occur in any order. Execution may continue as long as at least one transition is enabled.

In figure 1.1, we see a simple marked Petri net with five places and four transitions. In this example, transitions t_1 and t_2 are enabled; the marking can be represented as a vector $\{1,1,2,0,0\}$. If transition t_2 is fired, the new marking will be $\{1,1,1,1,0\}$ and transition t_4 will be enabled. This Petri net does not have parallel arcs; if, for example, there were two parallel arcs between p_3 and t_2, then firing t_2 would remove both tokens from p_3.

1.2.2 Analysis of Petri Nets

Petri nets are widely used in modeling; in particular, a Petri net may be used as a model of a concurrent system. For example, a network of communicating processors with shared memory or a communications protocol might be so modeled. For this model to be of use, it must be possible to analyze the model. The questions that one might ask about a

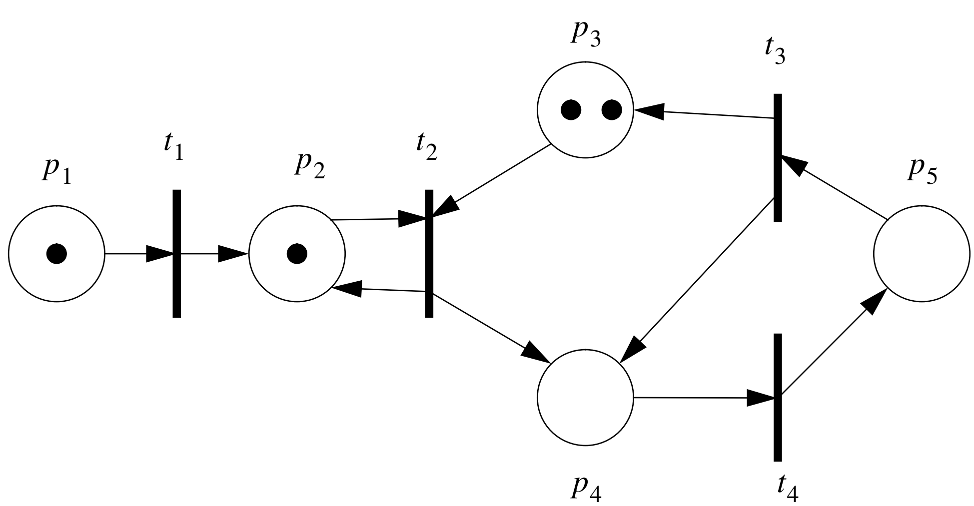


Figure 1.1 A simple Petri net.

Petri net model also apply when analyzing other models, both those that occur for models that are subsets of Petri nets and for other computational models that we will consider. The summary that follows is based on that of Peterson [Pet81] and Murata [Mur89].

For a Petri net to model a real hardware device, it is often necessary that the net have the property known as *safeness*. A Petri net with an initial marking μ is *safe* if it is not possible, by any sequence of transition firings, to reach a new marking μ' in which any place has more than one token. If this property is true, then a hardware model can represent a place as a single bit or, if the token represents data communication, space for a single datum.

It is possible to force a Petri net to be safe by adding arcs, provided that there are no parallel arcs connecting places and transitions. To force a place p_i to be safe, we add another place p'_i that has the property that p'_i has a token if and only if p_i does not have a token. To achieve this, transitions that use p_i are modified as follows [Pet81]:

If $p_i \in I(t_j)$ and $p_i \notin O(t_j)$, then add p'_i to $O(t_j)$.

If $p_i \in O(t_j)$ and $p_i \notin I(t_j)$, then add p'_i to $I(t_j)$.

This technique was used by Dennis to simplify the design of static dataflow machines [Den80]. In this context, these additional arcs are called *acknowledgment arcs*.

Safeness is a special case of a more general condition called *boundedness*. In many cases we do not require that the number of tokens in each place is limited to one; it will suffice to have a limit that can be computed in advance. A place is *k-bounded* if the number of tokens in that place never exceeds k, and a net as a whole is k-bounded if every place is k-bounded. If, for a Petri net, some k exists so that the net is k-bounded, we simply say that it is bounded. Where Petri nets are used as models of computation and tokens represent data, we can allocate static buffers to hold the data if the corresponding net is bounded.

Another important property of a Petri net model is *liveness*. Liveness is the avoidance of *deadlock*, a condition in which no transition may fire. Let $R(N, \mu)$ be the set of all markings that are reachable given the Petri net N with initial marking μ. Using the definition of [Com72], we say that a transition t_j is live if for each $\mu' \in R(N, \mu)$, there exists a sequence of legal transition executions σ such that t_j is enabled after that sequence is executed. Speaking informally, this means that no matter what transition sequence is executed, it is always possible to execute t_j again. A Petri net is live if every transition in it is live.[1]

Another important property of Petri net models is *conservativeness*; a net is *strictly conservative* if the number of tokens is never changed by any transition firing. A net is *conservative with respect to a weight vector w* if, for each place p_i we can find a weight w_i such that the weighted sum of tokens $\sum^{M} w_i \mu_i$ never changes; here μ_i is the number of tokens in the place p_i while the marking μ is in effect. Note that all Petri nets are conservative with respect to the all-zero vector. A net is said to be *conservative* (no modifiers) if it is conservative with respect to a weight vector with all elements greater than zero. Every conservative net is bounded, but not vice versa.

All the problems discussed so far are concerned with *reachable markings*, in the sense that they ask whether is it possible to reach a marking in which some property holds or does not hold. In that sense, given an algorithm for finding the structure of the set of reachable markings, we can answer these and other analysis questions.

The reachability tree represents the set of markings that may be reached from a particular initial marking for a given Petri net. The initial marking becomes the root node of the tree. Each node has one child node for each transition that is enabled by that marking; the tree is then recursively expanded, unless a node duplicates a node that was generated earlier. Note that if a net is *k*-bounded, for any *k*, this construction is finite; there are a

1. Commoner also defined lesser levels of liveness; this definition corresponds to “live at level 4”.

fixed number of distinct markings that are reachable from the initial marking. An additional rule is added to make the construction finite even for unbounded nets. To understand this construction, we define a partial ordering on markings. We say that $\mu' \geq \mu$ if, when considered as a vector, each element of μ' is greater than or equal to the corresponding element of μ (meaning that each place has as many or more tokens under marking μ' as under marking μ); we then say that $\mu' > \mu$ if and only if $\mu' \geq \mu$ and $\mu' \neq \mu$. Now consider a sequence of firings that starts at a marking μ and ends at a marking μ' such that $\mu' > \mu$. The new marking is the same as the initial marking except for extra tokens, so we could repeat the same firing sequence and generate a new firing μ'' that has even more tokens; in fact, when considered as a vector, $\mu'' - \mu' = \mu' - \mu$. Every place that gains tokens by this sequence of firings is unbounded; we can make its number of tokens grow arbitrarily large simply by repeating the firing sequence that changes the marking from μ to μ'. We represent the potentially infinite number of tokens associated with such places by a special symbol, ω, which can be thought of as representing infinity. When constructing the reachability tree, if we ever create a node whose marking is greater (in the sense we have just defined) than another node that occurs on the path between the root and the newly constructed node, we replace the elements that indicate the number of tokens in places that may grow arbitrarily large with ω. As we continue the construction of the tree, we assume that a place with ω tokens can have an arbitrary number of tokens added or removed and still have ω tokens. Given this convention, it can be shown that the resulting reachability tree (with infinitely growing chains of markings replaced by ω nodes) is finite for any Petri net; this construction and the proof was given by Karp and Miller [Kar69].

Given this construction, we have an algorithm for determining whether a Petri net is bounded: if the ω symbol does not appear in the reachability tree, the Petri net is bounded. Similarly, possible weight vectors for a conservativeness test can be determined by solving a system of *m* equations in *n* unknowns, where *m* is the number of nodes in the reachability tree and n is the number of places. These equations take the form

$$\mu_i^T w = 1 \tag{1-1}$$

where μ_i is the marking associated with the i^{th} node in the reachability graph, *and* w represents the unknown weight vector. We treat ω as representing an arbitrarily large number, so that any place that ever has a ω symbol must have zero weight. If the system is overly constrained there will be no nonzero solutions and the system will not be conservative. The reachability tree cannot be used to solve the liveness question if there is a ω entry, as this represents loss of information.

1.2.3 The Computation Graphs of Karp and Miller

The earliest reference to the dataflow paradigm as a model for computation appears to be the computation graphs of Karp and Miller [Kar66]. This model was designed to express parallel computation and represents the computation as a directed graph in which nodes represent an operation and arcs represent queues of data. Each node has associated with it a function for computing outputs from inputs. Furthermore, for each arc d_p, four nonnegative integers are associated with that arc:

A_p, the number of data words initially in the queue associated with the arc,

U_p, the number of data words added to the queue when the node that is connected to the input of the arc executes;

W_p, the number of data words removed from the queue when the node that is connected to the output of the arc executes;

T_p, a threshold giving the minimum queue length necessary for the output node to execute. We require $T_p \geq W_p$.

Karp and Miller prove that computation graphs with these properties are determinate; that is, the sequence of data values produced by each node does not depend on the order of execution of the actors, provided that the order of execution is valid. They also investigated the conditions that cause computations to terminate, while later views of data-flow computation usually seek conditions under which computations can proceed indefinitely (the avoidance of deadlock). They also give algorithms for determining storage requirements for each queue and for those queue lengths to remain bounded. In [Kar69], Karp and Miller extend this model to get a more general form called a “vector addition system”. In this model, for each actor we have a vector, and this vector represents the number of tokens to be added to each of a set of buffers. Negative-valued elements correspond to buffers from which tokens are subtracted if the actor executes. Actors may not execute if that would cause the number of tokens in some buffer to become negative. If the number of tokens in each buffer is represented as a vector, then executing an actor causes the vector for that actor to be added to the system state vector, hence the name “vector addition system.” If actors are identified with transitions and buffers are identified with places, we see that this model is equivalent to Petri nets.

It is not difficult to see that Karp and Miller’s computation graph model can be analyzed in terms of Petri nets. The queues of data can be modelled as places and the nodes can be modelled as transitions. Each arc of the computation graph can be modelled as U_p input arcs connecting a source transition to a place, followed by T_p output arcs connecting a place to an output transition and $T_p - W_p$ arcs connecting the output transition back to the place. The Petri net model differs from the computation graph model in that Petri net tokens do not convey information (other than by their presence or absence), only the number of tokens matters. Since Petri net tokens are all alike, the fact that streams of values are produced and consumed with a first-in first-out (FIFO) discipline is not reflected in the Petri net model. However, the constraints on the order of execution of transitions are exactly the same.

1.2.4 Marked Graphs

Marked graphs are a subclass of Petri nets. A marked graph is a Petri net in which every place has exactly one input transition and one output transition. Parallel arcs are not permitted. Because of this structural simplicity, we can represent a marked graph as a graph with only a single kind of node, corresponding to transitions, and consider the tokens to "live" on the arcs. This representation (with only one type of node corresponding to Petri net transitions) is standard in dataflow. Marked graphs can represent concurrency (corresponding to transitions that can be executed simultaneously) and synchronization (corresponding to multiple arcs coming together at a transition) but not conflict (in which the presence of a token permits the firing of any of several transitions, but firing any of the transitions disables the others). Marked graphs are much easier to analyze than general Petri nets; the properties of such graphs were first investigated in detail in [Com72].

In particular, the question of whether a marked graph is live or safe can be readily answered by looking at its cycles. A *cycle* of a marked graph is a closed sequence of transitions that form a directed loop in the graph. That is, each transition in the sequence has an output place that is also an input place for the next transition of the sequence, and the last transition in the sequence has an output place that is an input place for the first transition in the sequence. It is easy to see that if a transition that is in a cycle fires, the total number of tokens in the cycle will not change (one token is removed from an input place in the cycle and one is added to an output place in the cycle). From this it can be shown that:

- A marking on a marked graph is live if and only if the number of tokens on each cycle is at least one.
- A live marking is safe if and only if every place is in a cycle, and every cycle has exactly one token.

1.2.5 Homogeneous Dataflow Graphs

It is natural to model computation with marked graphs. We can consider transitions to model arithmetic operations; if we then constrain the graph to be safe, using the results just described, it is then possible to avoid queuing; each arc needs to store only a single datum. However, since it was shown earlier that it is possible to transform any ordinary marked Petri net into a safe net by the addition of acknowledgment arcs, it is usual to represent computation in terms of dataflow graphs without these extra arcs. The acknowledgment arcs may then be added, or we may execute the graph as if they were there (as in Petri's original model, in which a transition was not permitted to fire if an output place had a token). It is then necessary only to be sure that the resulting graph does not deadlock, which can only occur if there is a cycle of nodes (transitions) that does not contain a token.

The static dataflow model of Dennis was designed to work in this way: ideally, the rule was that a node could be evaluated as soon as tokens were present on all of its input arcs and no tokens were present on any of its output arcs. Instead, acknowledgment arcs were added, so that a node could be enabled as soon as tokens were present on all input arcs (including acknowledgment arcs) [Den80].

Dataflow actors that consume one token from each input arc and produce one token on each output arc are called *homogeneous*. The value, if any, of a token does not affect the eligibility of an actor to execute (though it usually does affect the value of the tokens computed). These restrictions are relaxed in more general dataflow models. Graphs consisting only of homogenous dataflow actors are called homogenous dataflow graphs and correspond to marked graphs.

Static dataflow machines permit actors other than homogeneous dataflow actors, such as the SWITCH and SELECT actors we will discuss in the next section. However, the constructs in which these actors appear must be carefully controlled in order to avoid deadlock given the constraint of one token per arc [Den75b].

1.2.6 General Dataflow Graphs

In the most general sense, a dataflow graph is a directed graph with actors represented by nodes and arcs representing connections between the actors. These connections convey values, corresponding to the tokens of Petri nets, between the nodes. Connections are conceptually FIFO queues, although as we will see, mechanisms are commonly used

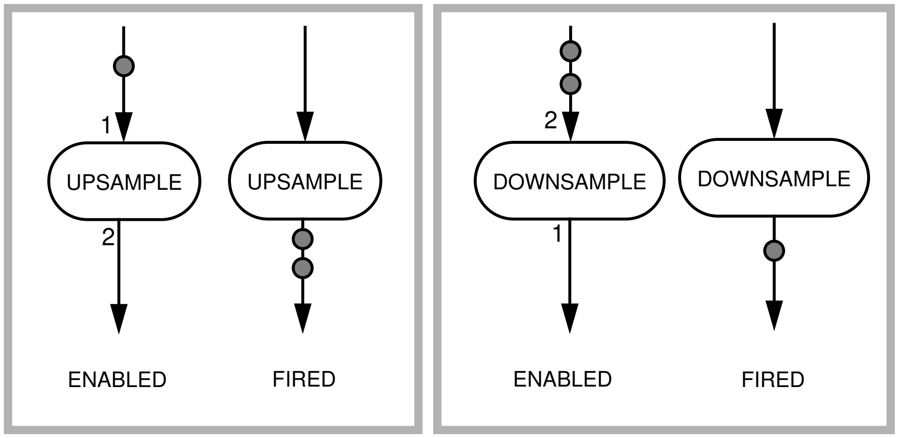


Figure 1.2 Regular dataflow actors produce and consume fixed numbers of tokens.

that permit out-of-order execution while preserving the semantics of FIFO connections. We permit initial tokens on arcs just as Petri nets have initial markings.[1]

If actors are permitted to produce and consume more than one actor per execution, but this number is constant and known, we obtain the synchronous[2] dataflow model of Lee and Messerschmitt [Lee87b]. We will call actors that produce and consume a constant number of tokens *regular actors*, and dataflow graphs that contain only regular actors *regular dataflow graphs*. The canonical non-homogeneous regular dataflow actors are UPSAMPLE and DOWNSAMPLE, shown in figure 1.2.

If no restrictions are made on when actors can fire other than data availability, the regular dataflow model is a subclass of Petri nets; it is obtained by starting with marked graphs and permitting parallel arcs between places and transitions, imposing the requirement that each place have only a single input transition and a single output transition. Lee's model is not, in fact, the same as this subclass of Petri nets because the execution sequence is chosen to have certain desirable properties, while Petri net transitions are per-

1. Ashcroft and Wadge [Ash75] would call this model "pipeline dataflow" and argue for a more general model, permitting data values to flow in both directions and not requiring FIFO, as in their Lucid language (see section 1.4.1). Theirs is a minority view; Caspi, for example [Cas92] contends that the Lucid model is not dataflow at all.
2. The term *synchronous* has been used in very different senses by Lee and by the designers of the stream languages LUSTRE [Hal91] and SIGNAL [Ben90]. We will use the term *regular* to refer to actors with constant input/output behavior to avoid this possible source of confusion.

mitted to fire whenever enabled. We will investigate the properties of Lee's model in detail in section 2.2.

We will use the term *dynamic actor* to describe a dataflow actor in which the number of tokens produced or consumed on one or more arcs is not a constant. As a rule, in such actors the numbers of tokens produced or consumed depends on the values of certain input tokens. These models are usually more powerful than Petri net models, as Petri net models are not Turing-equivalent, but, as we shall see, dynamic dataflow models usually are. However, this increase in expressive power also makes dynamic dataflow graphs much harder to analyze, as many analysis problems become undecidable.

We can conceive of actors whose token consumption and token production depends on the values of control inputs. The canonical examples of this type of actor are SWITCH and SELECT, whose function is shown in figure 1.3. The SWITCH actor consumes an input token and a control token. If the control token is TRUE, the input token is copied to the output labeled T; otherwise it is copied to the output labeled F. The SELECT actor performs the inverse operation, reading a token from the T input if the control token is TRUE, otherwise reading from the F input, and copying the token to the output. These actors are minor variants of the original Dennis actors [Den75b], are also used in [Wen75],

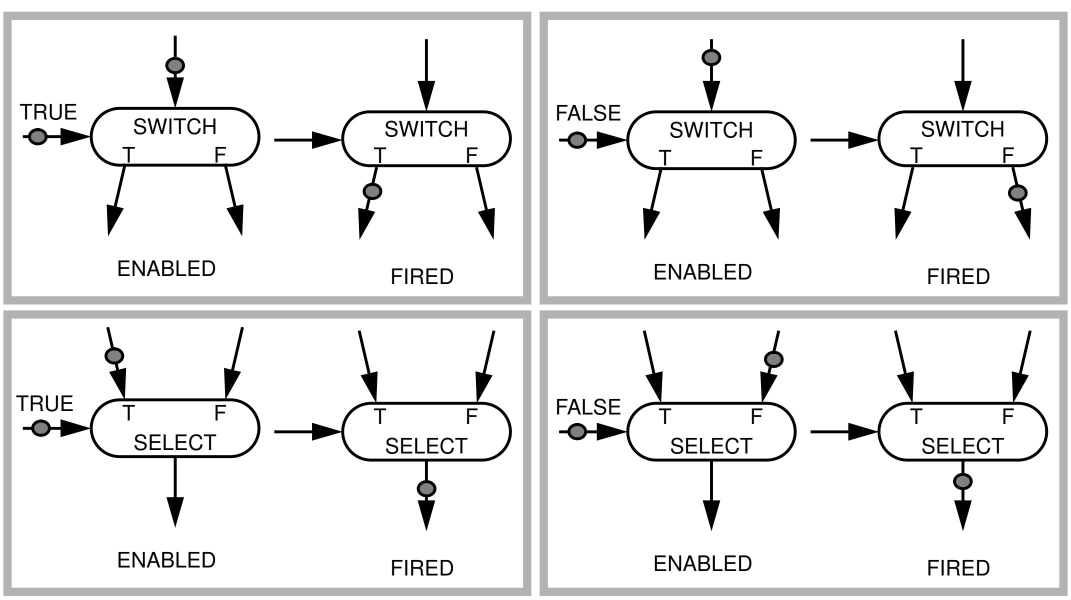


Figure 1.3 The dynamic dataflow actors SWITCH and SELECT.

[Tur81], and [Pin85], and are the same as the DISTRIBUTOR and SELECTOR actors in [Div82].

We can also conceive of actors whose behavior depends upon the timing of token arrivals. An example of this class of actor is the non-determinate merge actor, which passes tokens from its inputs to its output based on the order of arrival. This actor resembles the SELECT actor in the figure below except for the lack of a control input. Non-determinate actors may be desirable to permit dataflow programs to interact with multiple external events [Kos78]. In addition, if the set of admissible graphs is severely restricted, graphs with the nondeterminate merge can have a completely deterministic execution; for example, it can be used to construct the "well-behaved" dataflow schema discussed by Gao *et al* in [Gao92].

If the operations represented by the nodes of a dataflow graph are purely functional, we have a completely definitional model of computation. Some non-functional operations, such as those with history sensitivity, can also be accommodated within a definitional model; any dataflow actor that has state may be converted into an equivalent dataflow actor without state by the addition of a self-loop. The new actor accepts data inputs and a state input, and computes data outputs and a new state; the initial token value on the self-loop represents the initial state. If actors with state are represented in this manner, then dataflow programming strongly resembles functional programming, in that state is represented explicitly in arguments to functions and is explicitly passed around as an argument.

1.2.7 Kahn's Model for Parallel Computation

Kahn's small but very influential paper [Kah74] described the semantics for a language consisting of communicating sequential processes connected by sequential streams of data, which are produced and consumed in first-in first-out order. The model of computation is further developed in [Kah77]. No communication path exists between the processes other than the data streams; other than that, no restriction is placed on the implementation of each process — an imperative language could be used, or the process could simply invoke a function on the inputs to produce the output and therefore be state-free. Each process is permitted to read from its inputs in arbitrary order, but it is not permitted to test an input for the presence of data; all reads must block until the request for data can be met. Thus the SWITCH and SELECT actors of the previous section could be implemented as Kahn actors, but not the non-deterministic merge, since it would be necessary to commit to reading either the first input or the second, which would cause inputs on the opposite channel to be ignored. It is shown that, given this restriction, every stream of data that forms a communication stream is determinate, meaning that its history depends only on the definitions of the processes and any parameters, and not on the order of computation of the processes.

The semantics of Kahn's parallel process networks are a strict superset of the models considered by many dataflow and stream languages, as well as hybrid systems that permit actors to be implemented using imperative languages or to have state. Hence, when we say that all language constructs in a dataflow or stream model obey the Kahn condition, we mean that the model can be implemented without requiring input tests on streams or non-blocking read operations and we then can be assured that all data streams are determinate.

1.3. DATAFLOW COMPUTING

Dataflow computing originated largely in the work of Dennis in the early 70s. The dataflow model of computer architecture was designed to enforce the ordering of instruction execution according to data dependencies, but to permit independent operations to execute in parallel. Synchronization is enforced at the instruction level.

There have been two major varieties of "pure" dataflow machines, static and tagged-token. In a static dataflow machine, memory for storing data on arcs is preassigned, and presence bits indicate whether data are present or absent. In a tagged-token dataflow machine, token memory is dynamically allocated, and tags indicate the context and role of a particular token.

1.3.1 Static Dataflow Machines

The earliest example of a static dataflow machine was Dennis's MIT static dataflow architecture [Den75a], although the first machine to actually be built was Davis' DDM1 [Dav78].

In a static dataflow machine, dataflow graphs are executed more or less directly, with nodes in the graph corresponding to basic arithmetic operations of the machine. Such graphs, where nodes represent low-level operations, are called *fine-grain* dataflow graphs, as opposed to *coarse-grain* dataflow graphs in which nodes perform more complex operations. The graph is represented internally as a collection of *activity templates*, one per node. Activity templates contain a code specifying what instruction is to be executed, slots for holding operand values, and destination address fields, referring to operand slots of subsequent activity templates that need to receive the result value [Arv91]. It is required that there never be more than one token per arc; acknowledgment arcs are added to achieve this, so that a node is enabled as soon as tokens are present on all arcs (including acknowledgment arcs).

The original MIT static dataflow architecture consists of a group of processing elements (PEs) connected by a communication network. A diagram showing a single processing element appears in figure 1.4. The Activity Store holds activity templates that have empty spaces in their operand field and are waiting for operand values to arrive. The Update Unit receives new tokens and associates them with the appropriate activity template; when a template has all necessary operands, the address of the template is entered into the Instruction Queue. The Fetch Unit uses this information to fetch activities and forward them to the appropriate Operation Unit to perform the operation. The result value is combined with the destination addresses to determine where to send the result, which may need to go to the Update Unit of the same PE or to that of a a different PE through the communications network [Den80], [Den91].

The requirement that there be only one token per arc, and that communication between actors be synchronized by acknowledgment arcs, tends to limit the parallelism that can be achieved substantially. If waves of data are pipelined through one copy of the code, the available parallelism is limited by the number of operators in the graph. An alternative solution is to use several copies of the machine code [Den91].

1.3.2 Tagged-Token Dataflow Machines

Tagged-token dataflow machines were created to overcome some of the shortcomings of static dataflow machines. The goal of such machines is to support the execution of loop iterations and function/procedure invocations in parallel; accordingly, recursion is supported directly on tagged-token dataflow machines, while on static dataflow machines it is not supported directly. To make this possible, data values are carried by tokens that include a three-part tag. The first field of the tag marks the context, corresponding to the current procedure invocation; the corresponding concept in a conventional processor executing an Algol-like language is the stack frame. The second field of the tag marks the iteration number, used when loop iterations are executed in parallel. The final field identifies the activity, corresponding to the appropriate node in the dataflow graph — this might be an instruction address in the physical machine [Arv91]. A node is then enabled as soon as tokens with identical tags are present at each of its input arcs; all three fields must match.

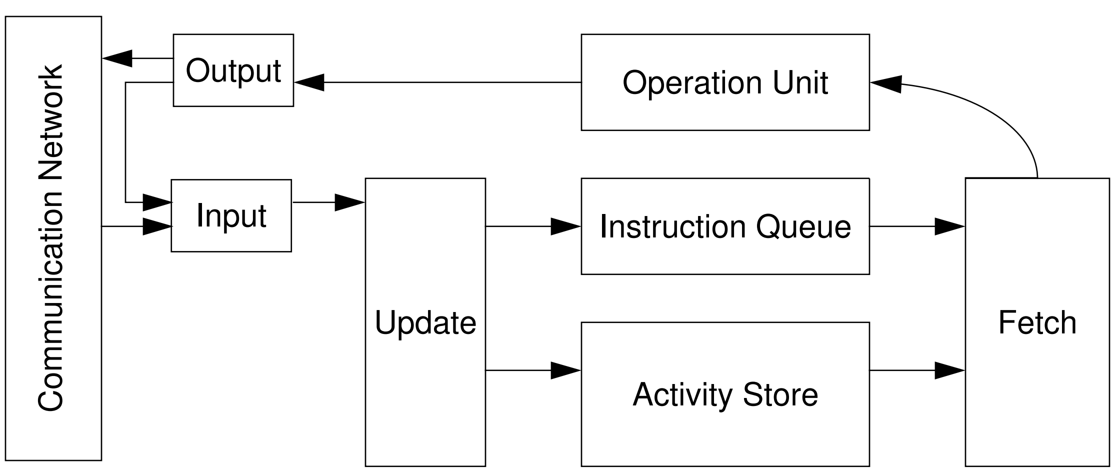


Figure 1.4 A simple model of a processing element for a static dataflow machine [Arv86]

No feedback signals (acknowledgment arcs) are required. A diagram of a single processing elemement of this type of machine appears in figure 1.5.

The MIT Tagged-Token Dataflow Machine [Arv90] and the Manchester Dataflow Computer [Gur85] were both independently designed according to the principles described above, roughly at the same time. The latter machine was actually built in 1981. In both machines, a “waiting-matching unit” is responsible for collecting tokens destined for binary operators and pairing them together, dispatching operations when a match is found. Unary operators may be dispatched immediately without going through the waiting-matching unit.

In addition to the structure described above, the MIT machine had a special type of storage for large data structures using the concept of *I-structures* [Arv90]. An I-structure is a composite object whose elements can each be written only once but can be read many times. These structures are *non-strict*, meaning that it is possible to perform an operation requiring some elements of the structure even though the computation of other elements of the structure is not yet complete. There are three operations defined on I-structures: *allocation*, which reserves a specified number of elements for the structure; *I-fetch*, which retrieves the content of a given element of the structure, deferring the operation if the element has not yet been computed, and *I-store,* which writes a given element of the struc-

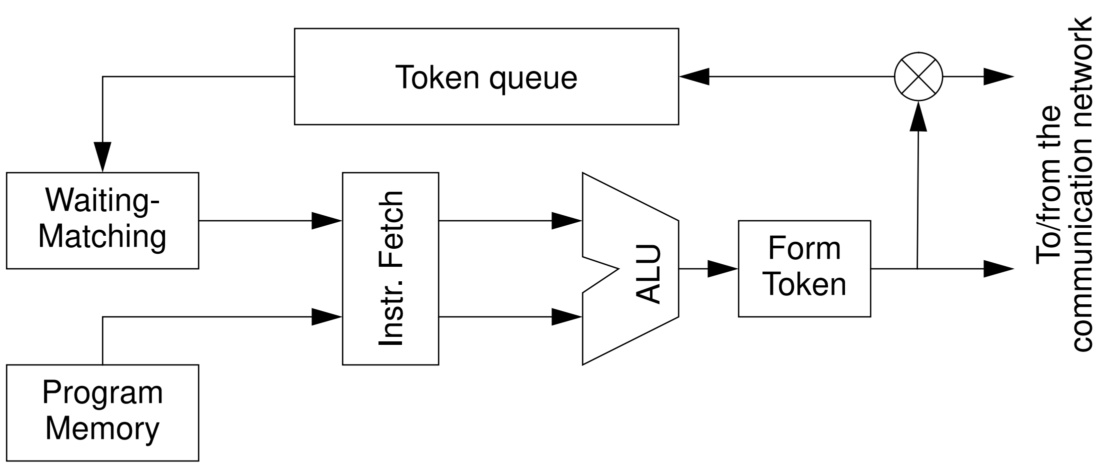


Figure 1.5 Block structure of a single processing element in the MIT tagged-token dataflow machine [Arv91].

ture, signalling an error if the element has already been written. The I-structure storage unit provides specialized hardware to support these rules, and tokens contain references to I-structures. I-structure operations are split-phase, meaning that the read request and the response to the request are two separate actions and do not cause the issuing processing element to wait.

One of the main problems with tagged-token machines has been that the waiting-matching unit is a bottleneck; the operation of matching the tokens is expensive and the amount of memory required to store tokens waiting for a match is large. A second problem is that the amount of parallelism that can be uncovered by the operation of a tagged-token machine is very large. If too many tokens are generated that must wait for a match and the waiting-matching unit fills with tokens, the machine deadlocks. Finally, the expensive token-matching functions are always performed, even on purely sequential code where they gain nothing because there is no parallelism to exploit.

Some of these problems have been addressed by subsequent architectural designs. For example, in the Monsoon project [Pap88], rather than allocating memory for tokens dynamically, explicitly addressed and statically allocated token store is used. In this model, a separate memory frame is allocated for each function activation and loop activation, much as a new stack frame is allocated on function entry on a conventional von Neumann machine that is executing an Algol-like language. To make this idea practical, we must limit the amount of parallelism in dataflow graphs (specifically, the number of loop iterations that may be active simultaneously) by means of special constructs. For this purpose, structures known as *k*-bounded loops were used [Cul89].

1.3.3 Dataflow/von Neumann Hybrid Machine Models

Dataflow machines were conceived of to address the problems of latency and synchronization, problems that have not been addressed as effectively as might be desired in von Neumann machines or in networks of such machines. Dataflow machines do synchronization on the execution of every fine-grain dataflow actor, at a smaller cost than would be required on a traditional processor. Unfortunately, on short segments of sequential code that have all required data in local high-speed storage (registers and cache), any overhead for synchronization is wasteful. These sequential code segments, corresponding to basic blocks operating on local variables in traditional imperative programming languages, are more efficiently executed by a RISC-style processor[1]. However, synchronization between processors is more efficiently handled using a dataflow approach. It therefore seems natural to attempt to combine the approaches.

The greatest deficiency of the pure dataflow model is the excessive token matching and overhead required for communication between actors. Enhancements that exploit temporal or spatial locality (caches, for example) are also hard to achieve in the pure dataflow model. Most of the hybrid models achieve a reduction in overhead by applying some form of clustering: certain sequences of actors are combined into threads, which are sequentially executed without incurring the cost of matching overhead.

Some of these hybrid approaches, such as [Bic91], retain the notion of the token and resemble traditional tagged-token machines, except for the clustering of actors into threads. Others, which have been described as "dataflow architectures without dataflow" [Gao88], retain a data-driven execution model but fetch all data from shared memory. A multilevel dataflow model, which exploits features of the von Neumann model such as virtual space, multilevel memory hierarchies, and RISC design principles, has been developed by Evripidou and Gaudiot [Evr91]; this project has some resemblance to that of Gao *et al*.

Finally, there is a category of machines that enhance RISC architecture with additional mechanisms for tolerating memory and communication latencies, supporting fine-grain synchronization among multiple threads of execution. MIT's Alewife project, using

1. A RISC (Reduced Instruction Set Complexity) processor, as used in most workstations today, is a pipelined von Neumann processor characterized by a load-store architecture, many general-purpose registers, a simple and regular instruction set, and a multilevel memory hierarchy including one or more caches [Hen90].

a modified form of the standard Sparc RISC architecture known as Sparcle, is the best known example [Aga93].

1.4. DATAFLOW AND STREAM LANGUAGES

Dataflow languages were first developed to support programming of dataflow machines. Since data dependencies were the organizing principle of the paradigm and since any artificial sequencing was objectionable, these languages were essentially functional languages. For several of the languages discussed, a user-written textual form is converted internally into a dataflow graph.

The two most important languages developed in the early days of dataflow machines were Val [Ack79], which later became Sisal, and Id [Arv82], which later became Id Nouveau [Nik86]. For the most part, these and other languages developed during that period did not have higher-order functions, and they were *strict* (meaning that all inputs to any function must be completely computed before the function can begin execution), reflecting the data-driven rather than demand-driven style of control used in dataflow machines (in which new data are produced as quickly as possible and constraints in the graphical structure are used as a throttling mechanism). Id also supports non-strict composite objects in the form of I-structures, whose semantics were discussed in section 1.3.2.

Another interesting and important dataflow language is Lucid [Ash75], which is distinguished by the use of identifiers to represent streams of values. A one-dimensional stream might represent a time series or a sequence of values passing through a dataflow node; Lucid also supports streams of higher dimension. This language was intended to have semantics that were sufficiently clear to prove assertions about parallel programs.

Finally, we will discuss the languages LUSTRE and SIGNAL, languages with a theoretical foundation that has contributed much to the solution of problems of consistency and boundedness in general dataflow.

1.4.1 Lucid

Lucid is a functional language in which every data object is a *stream* (a sequence of values). It is first-order: we may only construct new streams, not new functions. All Lucid operations map streams into streams. Like some of the other languages we will discuss in this section, it can be considered to be a dataflow language in which the variables (the streams) name the sequences of data values passing between the actors, which correspond to the functions and operators of the language. Skillcorn [Ski91] points out its resemblance to Kahn's networks of asynchronous processes [Kah74]; other stream languages, together with the graphical dataflow systems used in Gabriel [Bie90] and Ptolemy [Buc91], also fit this model. While Lucid supports multidimensional streams, we will discuss a subset of Lucid in which streams are one-dimensional and the elements of streams are either integers or Boolean-valued. We then have pointwise functions or operators, which construct new streams by applying sample by sample to existing operators. There are three special non-pointwise operators:

- **`initial`**, which takes a single stream argument and produces a new stream in which each element is equal to the first element of the input stream;
- **`succ`**, which takes a stream and discards the first element;
- **`cby`** (continued by), which is written as an infix operator, taking two streams. The output stream consists of the first element of the first stream argument, followed by the whole of the second argument.

There is also a pointwise conditional operator:

`if` ***`c`*** **`then`** ***`ts`*** **`else`** ***`fs`*** **(1–2)**

in which ***`c`*** is a Boolean stream and ***`ts`*** and ***`fs`*** are streams of the same type. This operator, if thought of as a dataflow actor, always consumes one element from each of the three input streams for each element produced in the output stream; this behavior is quite different from the behavior of conditionals in other stream languages, such as SIGNAL.

In addition, Lucid permits user-defined functions, which may be recursive.

As a simple example, a Lucid program (or definition, since Lucid is a definitional language) for the series of Fibonacci numbers, given in [Ski91] is

```
fib = 1 cby (1 cby (fib + succ fib))                    (1–3)
```

Parentheses have been added to make the structure of the program clearer. It is easy to see that the first two elements of **fib** are 1; in addition, it can be seen that element $n + 2$ is equal to the sum of elements n and $n + 1$.

Note that there is no way to subsample a stream using the above operators, meaning that we cannot produce a stream that has values "less frequently" than the input streams.

1.4.2 SISAL

SISAL is an acronym for "Streams and Iteration in a Single Assignment Language." SISAL originated in the dataflow community as the language Val [Ack89] and was used to program the Manchester Dataflow Machine [Gur85]. It has a target-architecture-independent dataflow graph intermediate form. The language has evolved into a complete functional language; for example, it has higher-order functions. Implementations exist for a variety of uniprocessors, shared-memory multiprocessors, the Cray X/MP, and other machines [Böh92]. It has been a major goal of the SISAL project to demonstrate sequential and parallel execution performance competitive with programs written in conventional languages, and impressive results have been achieved [Bur92].

SISAL has powerful features for manipulating arrays (including vector subscripts to select and manipulate subarrays) and non-strict stream types, which are produced in order by one expression evaluation and consumed in the same order by one or more other expression evaluations. As an example of a non-strict operation on streams consider the following, from a "Sieve of Eratosthenes" program:

```
function Sieve(S: stream[integer];
               M: integer returns stream[integer])
```

```
        for I in S returns
              stream of I unless mod(I,M) = 0
        end for
  end function
```

The above function accepts a stream of integers and produces another stream, and the result may be used before the stream is completely computed. Production and consumption of streams may be pipelined. Streams are usually generated by **for** expressions, as above.

There are two forms of **for** expressions. In the first form, values are distributed to (multiple instances of) the body of the **for** expression and each body instance contributes a value to the overall result (the result might be an array or stream, or a reduction operator might be applied). The **Sieve** function above has this type of **for** construct. In the second form, an iteration, dependencies are expressed between values defined in one body instance and values defined in the preceding body instance. Again, each body instance returns a value that contributes to the result. Here is an example of the iterative form:

```
  function Integers(lower: integer; upper: integer
              returns stream[integer])
        for initial
              I := lower;
        while I < upper repeat
              I := old I + 1;
        returns stream of I
        end for
  end function
```

This form of the **for** appears to have an imperative structure, but in fact does not; instead, we are defining the value that certain labels have in each body instance, and the relations between successive instances form a recurrence. It is not difficult to compile such recurrences into a dataflow graph intermediate form.

The program examples in this section are simplified versions of examples appearing in [Böh92].

1.4.3 SIGNAL and LUSTRE

SIGNAL and LUSTRE are both stream languages that owe part of their inspiration to Lucid. However, there are important differences between the approach used in these languages from the approach used in Lucid, and there is a sense in which these languages are much closer to what is usually meant by dataflow, although there are important distinctions, the main one being that queuing of values on arcs does not occur.[1] Both of these languages are descendants of ESTEREL [Ber92]. These languages form a family of tools for the design of *reactive systems*, including real-time systems and control automata. Time is explicitly modeled in all of these languages.

In Lucid, it is possible to define a stream so that "future" values depend on "past" values or vice versa, as long as there is some definition for each element. This is exploited effectively in [Ski91] for multidimensional cases in, for example, solving boundary value problems. In SIGNAL and LUSTRE, however, streams can be thought of as evolving in time, and operators that are not point-to-point are always causal (so that for each stream, "future" elements only depend upon "past" elements of the same and other streams). Furthermore, each stream variable has associated with it a clock, representing in an abstract sense the time instances at which a stream has values.

Like Lucid, in SIGNAL and LUSTRE streams can be constructed by applying pointwise operators to other streams, and there are constructs resembling Lucid's **succ** and **cby** operators. Conditional operators in these languages are quite different from Lucid, however; both SIGNAL and LUSTRE provide a **when** operator that has the effect of subsampling a stream, producing another stream that is "less frequent." For example, we could write

```
xp = x when x > 0                                   (1-4)
```

Having done this, we may inquire into the meaning of the statement

1. Differences between the synchronous model provided by these languages and the dataflow model are discussed in detail in section 2.3.5.

$$\mathbf{y = xp + x} \tag{1-5}$$

It appears that there is an inconsistency here; assuming that the stream **x** has both positive and negative values and that the stream is arriving at a steady rate, it appears that the two streams arriving to be summed have different sample rates (in that **xp** will contain fewer values than **x** in any given time interval). Both LUSTRE and SIGNAL use a mechanism called the "clock calculus" to determine whether it is valid to combine two streams in this manner. Due to some differences in the definitions of the two languages, there are some important differences in the clock calculus of the two languages. The clock calculus is discussed in detail in section 2.3.4.

The **when** operator can be thought of as representing one half of the SWITCH actor discussed in section 1.2.6 (there is a significant difference in that no queuing of tokens is permitted). One significant difference between the LUSTRE and the SIGNAL languages is what is done to replace the corresponding SELECT actor. The LUSTRE language has the **if/then/else** statement, with semantics like that of Lucid. This statement accepts a Boolean stream and two streams to be selected from. Just as for the dataflow SELECT actor, a token is consumed from the Boolean input stream for each output value produced (although it is not exactly the same as SELECT). Accordingly, this actor obeys the Kahn condition: it can be implemented by a communicating sequential process that never tests its inputs for the presence of data. Since other LUSTRE actors also obey the Kahn condition, all streams defined and computed by the language are deterministic. However, the **if/then/else** does not correspond to the dataflow SELECT, since all three input streams have the same rate in the LUSTRE model; a statement like

$$\mathbf{absx := if\ x > 0\ then\ x\ else\ -x} \tag{1-6}$$

would require both a SWITCH and a SELECT, or a conditional assignment, in a dataflow model.

SIGNAL provides a different actor to combine streams, **default**. This actor merges two streams to produce a third stream:

$$\texttt{a3 := a1 default a2} \qquad (1\text{–}7)$$

This actor produces a stream that is defined at any logical instant where at least one of the inputs **`a1`** or **`a2`** is defined; if both streams are defined at the same time, the value chosen is taken from the first argument, in this case **`a1`**. In [LeG91] this is called a deterministic merge, and indeed it is deterministic in the sense that, given a definition of the streams **`a1`** and **`a2`**, **`a3`** is always defined and comes out to the same answer. However, its lack of a control input makes it resemble the non-deterministic merge of dataflow. If the clocks of the two signals were given, indeed the operation would be deterministic, but in SIGNAL the definitions of the signals determine their clocks. The semantics of **`default`** permit the construction of non-deterministic systems, and they also violate the Kahn condition [Kah74] in that, if we attempt to implement the above statement by means of a process that reads streams **`a1`** and **`a2`** and outputs the stream **`a3`**, it cannot be done if we impose the restriction that read operations on input streams be non-blocking.

An example of a non-deterministic SIGNAL system can be found in [LeG91].

1.5. SUMMARY AND PREVIEW OF FUTURE CHAPTERS

This chapter has presented some of the basic models that are at the foundation of dataflow and functional models and attempted to place them in context, providing the basis for analytical models that will be presented in future chapters. Dataflow systems can be analyzed by considering the properties of the actors as communicating objects by building on Petri net theory, or by analyzing the properties of the streams of data that connect them, as is done in stream languages. For optimum performance, it is necessary to do as much work as possible at compile time, possibly by clustering the graph to find threads and allocating as many resources as possible in advance.

In the next chapter, we consider a very important special case of dataflow graphs: regular dataflow graphs, in which the entire computation can be scheduled at compile time. We then discuss attempts to extend this model to accommodate dynamic actors, and the “clock calculus” model of LUSTRE and SIGNAL will be developed in detail.

2

STATIC SCHEDULING OF DATAFLOW

Fallacy: It costs nothing to provide a level of functionality that exceeds what is required in the usual case.

—J. Hennessy & D. Patterson [Hen90]

Dataflow has been widely adopted as a model for digital signal processing (DSP) applications for two principal reasons. The first reason is that dataflow does not overly constrain the order of evaluation of the operations that make up the algorithm, permitting the available parallelism of the algorithm to be exploited. This advantage holds regardless of the application area. The second reason is that a graphical dataflow model, or the model provided by a stream language such as Lucid, frequently is an intuitive model for the way that DSP designers think about systems: operators act upon streams of data to produce additional streams of data. Accordingly, coarse-grain dataflow has been applied to DSP since the beginning, in the form of languages that directly execute block diagrams in some form. DSP researchers and users have found this kind of dataflow representation useful even when there is no possibility of exploiting parallelism (because the whole graph will be executed by a sequential processor, for example).

Digital signal processing differs from other application areas in that the amount of data-dependent decision making is small, the structures of the problems are regular, and applications typically have very tight constraints on cost, together with hard real time deadlines that must be met. Because design trade-offs are frequently very different from those common in the more "mainstream" computer market, the DSP community has its own programmable digital signal processors, languages, and software.

On the application of dataflow to DSP, Lee commented that "the dataflow techniques of general purpose computing are too expensive for DSP and more powerful than what is required" [Lee91a]. This is because many DSP algorithms have almost no decision making, meaning that large parts of the problem can be efficiently scheduled at compile time for single or multiple processors. Of course, "little decision making" is not the same as "none", and to forbid all data-dependent decision-making will prevent us from using some valuable algorithms. Nevertheless, we will begin our explorations of static scheduling of algorithms for DSP with the assumption that there is no data-dependent decision making at all, and then later relax this assumption.

2.1. COMPILE-TIME VERSUS RUN-TIME SCHEDULING

For the purposes of this thesis, we define scheduling to consist of three operations: assigning actors to processors, determining the order of execution of the actors on each processor, and determining the exact starting time of each actor. Every system that executes a dataflow graph must perform all of these tasks; however, depending on the implementation and on the information we have about the execution requirements of the graph, some functions may be performed at "compile time", leaving others to be performed at "run time."

In [Lee89], Lee introduces a "scheduling taxonomy" defining four classes of scheduling (see figure 2.1). He uses the term *fully dynamic* to describe implementations in which all decisions about the execution of the graph are deferred until run time. We may delay the assignment of an actor to a processor until its input data are ready, for example, and then choose the first available processor. Some dataflow machines, such as the original MIT static dataflow architecture [Den75a], used this style of execution.

It is also possible to partition the actors of the dataflow graph between the various processors in advance, but then have the processors determine the order of actor execution at run time; this is called *static allocation*. Many dataflow machines work this way, for example, the Monsoon architecture [Pap88]. In the third type of scheduling, the compiler determines both the processor assignment and the order of execution of each node, but does not determine the exact timing of actor execution; where inter-processor communication exists, implicit or explicit synchronization operations are required so that actors will wait for data to become available. This technique is commonly used when there is no hardware support for scheduling, as when generating code for networks of von Neumann processors with shared memory. The Gabriel system [Bie90] is one example of this. The final possibility is to make all decisions at compile time, and this is called *fully static* scheduling.

If we possess an accurate model of the execution requirements and data dependencies of every actor, together with the properties of the target architecture to be used

	assignment	ordering	timing
fully dynamic	RUN	RUN	RUN
static allocation	COMPILE	RUN	RUN
self-timed	COMPILE	COMPILE	RUN
fully static	COMPILE	COMPILE	COMPILE

Figure 2.1 The time which the scheduling activities “assignment”, “ordering”, and “timing” are performed is shown for four classes of schedulers. The scheduling activities are listed on top and the strategies on the left [Lee89].

(including the costs and restrictions on communication between processors), then it is possible in principle to construct an optimal schedule for the execution of a dataflow graph on a given system of parallel processors with no run-time overhead for scheduling purposes. In practice, the multi-processor scheduling problem is NP-complete even in the simplest cases (see [Sih91] for a detailed discussion of NP-completeness as it applies to multiprocessor scheduling) so that most researchers use heuristic methods to obtain near-optimal schedules with various definitions of "goodness." Many of these techniques are elaborations on Hu's list scheduling ([Hu61]). Nevertheless, some researchers have built systems that produce optimal static multiprocessor schedules for DSP systems for some special cases (for example, [Sch86] and [Gel93]).

Even when it is possible to generate a fully static schedule, it is sometimes preferable to produce code for a self-timed system anyway, because such a system is considerably more robust to variations in timing because of minor differences in clock rates, errors in the specifications for timing of some operations, interrupts, and other factors. As long as the generated code conforms to Kahn's model of communicating sequential processes [Kah74], the self-timed system will execute reliably regardless of variations in timing.

When dynamic actors (actors whose execution is data-dependent) are included in the dataflow graph, it is clear that at least some scheduling decisions must be made at run time. Nevertheless, many of the techniques used for compile-time scheduling can be modified so as to remain applicable on dynamic dataflow graphs; it is not necessary to switch to a fully dynamic execution model. These techniques form the core of this thesis.

2.2. SCHEDULING OF REGULAR DATAFLOW GRAPHS

If a dataflow graph contains only actors for which the number of tokens produced and consumed on each arc is known in advance, and the time required to execute each actor is known with precision, it is then possible in principle to produce a fully optimal multiprocessor schedule for that graph (as discussed in the previous section, we must often settle for a near-optimal schedule because of the computational complexity of the schedul-

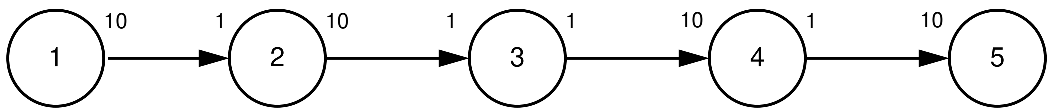


Figure 2.2 A regular dataflow graph. The numbers adjacent to arcs give the number of tokens produced or consumed on that arc by the associated actor.

ing problem). We will call dataflow actors with this property (known and constant numbers of tokens produced and consumed) regular dataflow actors, and graphs consisting only of regular actors will be called regular dataflow graphs.[1]

Clearly, regular dataflow graphs cannot have data-dependent firings of actors, as might occur in an if-then-else construct or a loop in which the number of iterations is determined by a computed value. But by imposing this limitation we obtain several very useful qualities: we can detect "sample rate inconsistencies" corresponding to unbounded numbers of tokens on arcs, or starvation conditions corresponding to deadlock. If these do not occur, a periodic schedule is always possible that permits the graph to be repeatedly executed on unbounded streams of data, and it is also possible to construct an acyclic precedence graph that permits the construction of a near-optimal multi-processor schedule. Finally, memory for data buffers between actors may be allocated statically, meaning that we are no longer constrained to FIFO processing of data streams in many cases, and that it is unnecessary to pay the overhead of a tagged-token system. That is, the compiler can associate static memory locations with actor firings to exploit data parallelism fully when there is no data dependency between successive firings of the same actor. Section 2.2.2 will demonstrate in detail how this is done.

2.2.1 The Balance Equations for a Regular Dataflow Graph

In figure 2.2 below we present a simple example of a regular dataflow graph. In order to produce a compile-time schedule for the repeated execution of this graph, it is first necessary to solve the *balance equations* for the graph, which determine the relative num-

1. This terminology is from [Gao92]; Lee used the term "synchronous data flow" [Lee87b] but this can be confused with the use of the term "synchronous" for the LUSTRE model [Hal91].

ber of iterations for each actor that will ensure that the number of tokens produced on each arc is equal to the number of tokens consumed. This produces one equation to be solved for each arc. It is convenient to express the resulting equations in matrix form; to do so, we define the *topology matrix* Γ. This matrix has one row for each actor in the graph and one column for each arc; the element γ_{ij} represents the number of tokens added to arc j by the execution of actor i. If the arc is an input arc for the actor, the value of the corresponding element in the topology matrix will be negative.

We now wish to find a *repetition vector* $\vec{r}$, whose i^{th} element represents the number of times to execute actor i, that solves the equation

$$\Gamma\vec{r} = \vec{o} \tag{2-1}$$

where $\vec{o}$ is the zero vector. For example, given the graph in figure 2.2, the topology matrix is

$$\Gamma = \begin{bmatrix} 10 & -1 & 0 & 0 & 0 \\ 0 & 10 & -1 & 0 & 0 \\ 0 & 0 & 1 & -10 & 0 \\ 0 & 0 & 0 & 1 & -10 \end{bmatrix}. \tag{2-2}$$

It can be seen that all solutions to the equation are of the form

$$\vec{r} = k\begin{bmatrix} 1 \\ 10 \\ 100 \\ 10 \\ 1 \end{bmatrix} \tag{2-3}$$

where k is arbitrary and the smallest integer solution has $k = 1$. It is shown in [Lee87b] that a necessary condition for the existence of a periodic schedule for a connected regular dataflow graph is that the rank of Γ be equal to one less than the number of actors, or equivalently, that the null space have dimension 1. For a collection of disconnected graphs, the null space must have dimension equal to the number of disconnected graphs, and the problem can be decomposed into separate systems of equations for each of the dis-

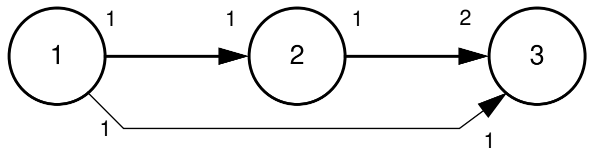


Figure 2.3 An inconsistent regular dataflow graph.

connected graphs.

If there is no solution to (2-1) except for the zero vector, we say that the graph is *inconsistent*. Inconsistency occurs if and only if there is an undirected cycle of arcs in the graph that is inconsistent in the following sense: treat the graph as an non-directed graph for the purpose of the consistency check; then any loop of arcs may be considered a cycle, regardless of the direction of the arrows. Consider a sequence of arcs $e_0, e_1, \ldots, e_{n-1}$ that form such a loop. Let a_0 designate the starting actor, which is connected to arc e_0 and arc e_{n-1}, and let actor a_i be the actor that is connected to arcs e_{i-1} and e_i. We now define the *gain* g_i of an arc e_i to be equal to the ratio of the number of tokens produced or consumed by actor a_i on arc e_i, divided by the number of tokens produced or consumed by actor a_{i-1} on the same arc (actor a_{-1} is identified with a_{n-1}). The cycle is inconsistent if the following condition does not hold:

$$\prod_{i=0}^{n-1} g_i = 1 \tag{2-4}$$

That is, the gain around every undirected cycle must be equal to one. As an example, for the graph in figure 2.3, we obtain a product of 2 and therefore the graph is inconsistent.

This result is easily proved by considering the following algorithm for solving for the repetitions vector: arbitrarily choose an actor and set its repetitions value to one. Next, for each arc connected to that actor, set the repetitions value of the adjacent actor (the actor connected to the opposite end of the arc) to the appropriate value to solve the balance equation for the arc; that is, if arc i connects actor j and actor k, then we must have

$$r_j \gamma_{ji} = -r_k \gamma_{ki} \tag{2-5}$$

where the γ terms are the elements of the topology matrix Γ. This algorithm is applied iteratively until all the *r* values are set. If the graph contains cycles, then the algorithm will visit some nodes more than once; in this case, a consistency check is performed; if the newly computed value for r_j differs from its previously reported value, inconsistency is reported. It is easy to see that there will always be inconsistency if there exists a cycle where the product of gains around the loop is not one. If there is no inconsistency, and any of the r_j values are fractional, the terms are multiplied by the least common multiple of the denominators to obtain the smallest integer repetition vector.

2.2.2 From the Balance Equations to the Schedule

Given an integer solution for the repetitions vector, it can be seen that if each actor is executed the number of times specified in its element of the repetitions vector, the graph will return to its original state, because the repetitions vector is in the null space of the topology matrix. However, it may not be possible to find a valid schedule with this number of iterations if the graph deadlocks. Deadlock occurs if there are too few initial tokens in a directed cycle of the graph to permit continued execution of that cycle. One simple algorithm for determining whether the graph deadlocks is to simulate the execution of the graph on a single processor: we execute enabled actors (source actors or actors with sufficient input tokens) until each actor has been executed the number of times specified in the repetitions vector, or until it is not possible to execute any actor. If we succeed in executing each actor the correct number of times, we know that deadlock does not occur and we also have one possible single-processor schedule.

If we wish to schedule the graph for execution on multiple processors, we then construct the acyclic precedence graph (APG) corresponding to the dataflow graph. The APG can be thought of as a model of the parallel execution of the dataflow graph on an unlimited number of parallel processors. Each node in the APG corresponds to a single execution of an actor in the original dataflow graph.

The graph is constructed as follows: first, we find the repetition vector to determine the required number of executions of each actor. All required actor executions that can be accomplished because the actors are source nodes, or because there are sufficient initial tokens to permit execution of the actors, are added to the structure as root notes. In figure 2.4 below, actor A must be executed twice, and both executions can be accomplished immediately. Executing actors makes it possible to execute other actors, so we add nodes corresponding to the execution of actors to the graph, adding arcs representing data dependencies, continuing until the number of executions of each actor corresponds to the repetition vector. In figure 2.4, the APG is completed by adding nodes for the three executions of actor B with arcs corresponding to the data dependencies. Since precedence in a dataflow graph is determined solely by data dependency, arcs in an APG imply precedence as well as data communication.

A more systematic way to produce the APG is to first transform the original regular dataflow graph into a homogeneous dataflow graph, using the procedure described in [Lee87b]. Next, arcs containing initial tokens are converted into a pair of input and output nodes. The output node is connected to the source actor of the removed arc, and the input node is connected to the destination actor of the removed arc. A unified algorithm for expansion of the graph to the homogeneous form together with construction of the APG is given in an appendix of [Sih91].

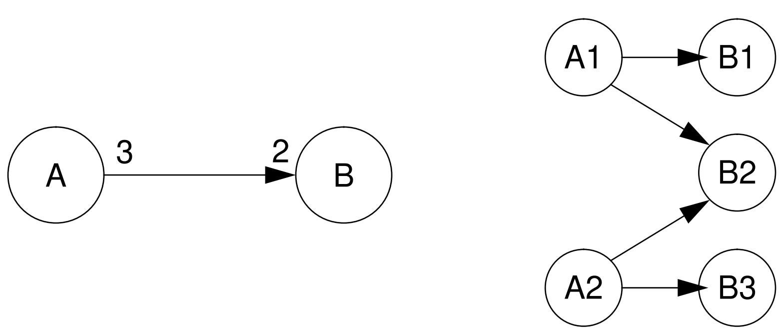


Figure 2.4 A simple regular dataflow graph and its associated acyclic precedence graph. Numbers adjacent to arcs specify the numbers of tokens produced and consumed.

Given a specific schedule for a regular dataflow graph, memory requirements for each arc may be determined and memory may be allocated in a static manner. This static allocation permits the execution of the graph to be performed out of order to some extent, much as in a tagged-token dataflow machine. For example, in figure 2.4 the executions of actors A and B may be done in parallel wherever there are no arcs specifying a data dependency.

When scheduling a dataflow graph for multiple processors, we may choose to minimize the *makespan*, which is the time for executing a single repetition of the graph. However, as a rule in DSP applications, the graph is repeatedly executed on a conceptually infinite input data stream, so a more reasonable objective is to minimize the iteration period, implicitly permitting a pipelined schedule. A third alternative is to construct a blocked schedule that executes the graph *k* times, for some *k*. Scheduling criteria are discussed extensively in [Ha92], [Sih91], and [Hoa93].

2.2.3 Comparison With Petri Net Models

Regular dataflow graphs can be considered as a special case of Petri nets, where actors become transitions and arcs become places, and there are multiple arcs connecting transitions and places corresponding to the number of tokens produced and consumed by each actor. There is, however, an important distinction in the analysis. A Petri net model is considered bounded if it is not possible for the number of tokens in a place to exceed the bound; however, because we schedule the execution of regular dataflow graphs at compile time, we do not need so strong a property; it is enough that schedules exist that yield bounded numbers of tokens on arcs. The values of the bounds no longer depend only on the topology of the graph; they also depend on the schedule chosen. For example, consider the graph in figure 2.4, and assume that we wish to schedule the graph on a single processor. If the schedule ABABB is chosen, the maximum size of the token buffer between actors A and B is four tokens. If, on the other hand, the schedule AABBB is chosen, the maximum buffer size is six tokens.[1] In addition, if the graph is to be executed repeatedly, schedules like 100A,150B are admissible, giving a much larger buffer size. Computed, schedule-dependent bounds such as these can be turned into "topological bounds" (bounds that are properties of the graph itself, as in bounded Petri nets) by adding acknowledgment arcs, and this is the procedure normally used to prepare graphs for execution on static dataflow machines [Den75a]. These arcs may limit parallelism; in fact, with tagged-token dataflow machines this limitation on parallelism is usually deliberate, to keep the machine from saturating [Cul89].

There is an interesting connection between the condition for a regular dataflow graph to be consistent and the condition for a Petri net to be conservative with respect to a nonzero weight vector. The latter condition requires that for each place p_i (corresponding in a regular dataflow graph to an arc) we can choose a fixed weight w_i such that the weighted sum of tokens in the graph does not change by the execution of any transition. The former condition (consistency of a regular dataflow graph) requires that the system of equations

$$\Gamma\vec{r} = \vec{o} \tag{2-6}$$

have a nontrivial solution. But the Petri new condition for a weight vector is simply

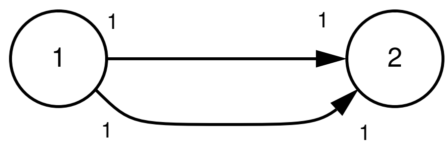
$$\Gamma^T\vec{w} = \vec{o} \tag{2-7}$$

which is precisely the dual of the previous equation (the dual of a Petri net is formed by replacing transitions with places and vice versa, which replaces the topology matrix by its transpose).

Unfortunately no generally useful results (that the author is familiar with) have yet been obtained from this observation. Some simple results can be obtained: if there are at least as many arcs as actors, the null space of Γ^T will have a dimension at least as high as that of Γ, so that a consistent dataflow graph will be conservative with respect to a weight

1. The latter schedule might be preferable in a compilation environment because the code to execute 2(A)3(B) would be more compact. Generation of compact looped schedules is discussed in [Bha93a].

vector for tokens $\vec{w}$ that is not all zeros. But it is possible that even so, there is no solution for which all weights are positive, and Petri nets for which negative weights must be assigned to some places are not considered conservative. For example, the graph below is conservative with respect to weight vectors of the form $\begin{bmatrix} -k & k \end{bmatrix}$, but is not conservative:



2.2.4 Limitations of Regular Dataflow Graphs

Regular dataflow graphs successfully represent unconditionally executed sequences of computation, and they also represent iteration successfully in situations where the number of executions is known and independent of the data. Conditional execution and data-dependent iteration are not represented, and neither is recursion.

However, in limited circumstances regular dataflow techniques can still be used when there is some conditional execution or when recursion takes a simple form. In many cases tail recursion can be transformed into a recurrence, which can be represented as feedback paths in a dataflow graph [Lee88a]. Also, it is sometimes suitable to replace conditional execution with conditional assignment. In conditional assignment, both alternatives of a conditional expression are computed, but one is discarded. This is an efficient approach when the cost of evaluating the expressions is small. In a hard real-time environment, it may also make sense to use conditional assignment if only one alternative of the conditional expression is expensive to compute, since we must allow time for the more expensive computation in order to assure that the deadline can be met. For these reasons, and because of the expense of conditional branches in pipelined processors, many DSP and RISC architectures have a conditional assignment operation. However, if both alternatives are expensive, then regular dataflow techniques are no longer sufficient for a good solution.

2.3. EXTENDING THE REGULAR DATAFLOW MODEL

Because of the powerful techniques that are available for the analysis of dataflow graphs, whether homogeneous or regular, and because they cannot solve all problems, it is only natural that they have been extended in a variety of ways to solve a larger class of problems. In comparing these models, there are a variety of considerations that might be applied:

Expressive power. Some extended models are equivalent in expressive power to Turing machines, which, as far as we know, means that they may represent any computable function[1]. Others are less powerful, while still being more expressive than regular dataflow graphs.

Compactness. A change in the properties of a model may not increase expressive power but may permit much more compact representation; the generalization from homogeneous dataflow graphs to regular dataflow graphs is an example of this.

Ease of analysis. Some types of models are easier to analyze than others. As a rule, ease of analysis and expressive power are in competition; many analysis questions are, in fact, undecidable for models that are equivalent to Turing machines, since they are equivalent to the halting problem.

Intuitive appeal. When a model is used in a particular application, it helps if the concepts in the model are closely related to concepts in the physical system being modeled.

2.3.1 Control Flow/Dataflow Hybrid Models

In a hybrid control flow/data flow model, control dependency and data dependency are combined. Such models normally imply a sequential mode of computation while permitting some freedom for re-ordering computation. A node in such a structure may have arcs that imply the communication of data as well as arcs that model the flow of control.

1. The assertion that no model of computation can compute a function that a Turing machine cannot compute is equivalent to Church's thesis [Chu32], which made similar statements about the (equally expressive) lambda calculus.

These models are used widely in compilers for traditional imperative high level languages. As a rule, a basic block in such a language is modeled as an acyclic homogeneous dataflow graph, and this graph, in turn, is a single node in a directed graph modeling the control flow structure. In Aho, Sethi, and Ullman [Aho86], the inner structure is called a *directed acyclic graph* or *dag*, and the outer structure is called a *flow graph*. An optimizing compiler (a misnomer, but a standard one) analyzes this structure to collect information about the program as a whole, permitting dead code elimination, restructuring to improve performance (moving invariant code out of loops or elimination of induction variables, for example), and allocation of program constructs to registers.

This model is an intuitive internal representation for a program written in an imperative, sequential language (such as Fortran, C, or Pascal) because it reflects its structure closely; the flow of control in the flow graph represents the flow of control specified by the user. Furthermore, there are over two decades of extensive experience with the analysis of this type of structure; chapter 10 of [Aho86] provides an extensive bibliography. Assuming typical underlying primitives, the model is also Turing equivalent. Given these advantages, we can expect this sort of structure to be used for a long time to come. However, there are significant disadvantages, caused mainly by the close association with the operational, imperative model of programming implied. Furthermore, the model is inherently sequential, although analysis might be able to uncover some parallelism.

The above model has two levels, with control flow at the top level and data flow underneath. There are also modeling techniques that combine the two levels, permitting both control flow and data flow at the same level. The dataflow/event graphs of [Whi92] are one such model; the PFG graphical programming language [Sto88] provides a similar capability.

The Program Dependence Web, or PDW, is a relatively new program representation for use as an intermediate representation in compilation of imperative languages [Bal90]. This model, an extension of program dependence graphs (PDG) [Fer87] and static single assignment (SSA) form [Cyt89], is designed to support mapping of conventional imperative languages onto dataflow architectures and includes all the necessary information to permit either a control-driven, data-driven, or demand-driven interpretation, a feature the PDG and SSA forms lack. The structure is naturally interpreted as a dataflow graph with controlled use of dynamic dataflow actors to assure that arcs never have more than one token, although other interpretations are also possible. It differs from the structure we will discuss in the next section in that a greater variety of dynamic actors are used and that initial tokens on arcs are not used.

2.3.2 Controlled Use of Dynamic Dataflow Actors

While there may be advantages to representing control flow and data flow separately as in the previous section, there is much to be said for a unified model in which dataflow is used throughout. Such models utilize dynamic dataflow actors (see section 1.2.6), with the consequence that the number of tokens produced or consumed on one or more arcs of the graph is determined only at “run time.” Once dynamic dataflow actors are permitted, new problems arise; it is difficult to assure consistency of flow rates, and as we shall see, certain analysis questions (such as whether the graph can be scheduled to require bounded memory) become undecidable if no restrictions are placed on the use of dynamic actors.

Nevertheless, dynamic actors have been in use for a long time, since the early work on static dataflow machines. Analysis problems are avoided by restricting the contexts in which they appear. Thus the fundamental distinction of approaches described in this section is that dataflow graphs are built up out of regular actors and specially restricted clusters of actors known as *schema* that behave, when taken together, as regular actors. The resulting graphs have many of the same attractive properties as graphs composed only of regular actors; accordingly, Gao *et al.*, who advocate this approach, call such graphs *well-behaved* [Gao92].

Consider the system in figure 2.5. This is an example of the standard "conditional schema," in which either actor 3 or actor 4 is conditionally executed based on the control token produced by actor 7, using a data token from actor 1 as an input. The result is sent to actor 6. We make the observation that the circled subsystem, including actors 2, 3, 4, and 5, can itself be treated as a regular dataflow actor which, on each execution, consumes a single token from each of two inputs and produces a single token on its output. When considered as a coarse-grain dataflow actor so that the cluster as a whole becomes an actor, we again have a regular dataflow system. Furthermore, it is easy to arrange the scheduling so that no arc ever contains more than a single token.

Instead of using the SELECT actor, it would be possible to replace it by the non-deterministic merge actor (which differs from SELECT in that it has no control input but simply transfers tokens from either data input onto its data output). If used in this context, and if executed as soon as an input token appears on either input, the graph as a whole has deterministic behavior despite the presence of a non-deterministic actor. This property is used, for example, in the program dependence web model of Ballance *et al.* [Bal90].

In this example, actors 3 and 4 have one input and one output. We can construct other conditional schema in which the actors corresponding to 3 and 4 have m inputs and n outputs each, for any m and n (each actor is assumed to have the same interface), together with a network of SWITCH and SELECT actors to route inputs appropriately. The result-

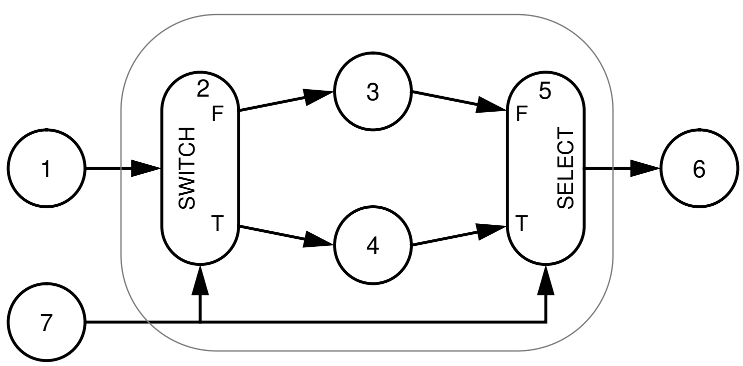


Figure 2.5 A dataflow graph with a "conditional schema." The numbers on the actors identify them. We consider the circled actors (2, 3, 4, and 5) as a subsystem. All actors other than SWITCH and SELECT are homogeneous.

ing system will then look like a homogeneous dataflow actor with $m+1$ inputs (including the control input) and n outputs. Again, with a suitable scheduling discipline it is possible to replace the SELECT actors with non-deterministic merge actors without loss of determinism.

Similarly, it is possible to construct well-behaved standard schemas for data-dependent iteration. It is useful to divide data-dependent iteration into two cases: convergent iteration, in which the condition for termination of the iteration is determined by the data computed by the most recent iteration, and iteration in which the number of iterations is known before the iteration starts (but not at compile time). It is usually possible to exploit more parallelism in the latter case. Examples of these types of constructs will be studied in detail in chapter 3.

2.3.3 Quasi-static Scheduling of Dynamic Constructs for Multiple Processors

If the language of regular dataflow graphs is extended to permit the conditional and loop schema described above, but no other uses of dynamic actors, additional complications either for dynamic execution of the graph (for example, by a dataflow machine or a simulator for such a machine) or for compile-time scheduling for a single processor are minimal. All that is necessary is to execute, or generate code for, a conditional branch or loop. For compile-time scheduling for parallel processors, more is required.

In [Lee88a], Lee proposed a technique called *quasi-static scheduling*, in which some actor firing decisions are made at run time, but only where absolutely necessary. Consider the system in figure 2.6, taken from [Lee88a]. In this case, we wish to schedule the execution of the system onto three sequential processors. The Gantt charts show the activity of the processors for two possible outcomes: in the first Gantt chart, the control token is TRUE, and the schedule includes the execution of the TRUE subgraph. The second chart shows the execution of the FALSE subgraph. Lee's key contribution was to note that, if idle times are inserted into both schedules so that the pattern of processor availability is the same regardless of the outcome of the conditional, static scheduling can then proceed after the execution of the conditional construct exactly as if it were a regular subgraph. This padding is required for fully static scheduling; if synchronization is used for data communication between processors the padding can be eliminated. Lee proposes a recursive structure for the scheduler that permits nested conditionals to be handled cleanly, and Ha extends and generalizes this idea ([Ha91], [Ha92]).

Depending on whether the goal is to meet hard real-time deadlines or to minimize the expected completion time, different scheduling strategies are appropriate. For a hard real-time system, it is advantageous to minimize the maximum time required; if the probability distribution of the Boolean control stream is known, it may be possible to minimize the expected time to completion instead.

The same essential idea (create a schedule in which the pattern of processor avail-

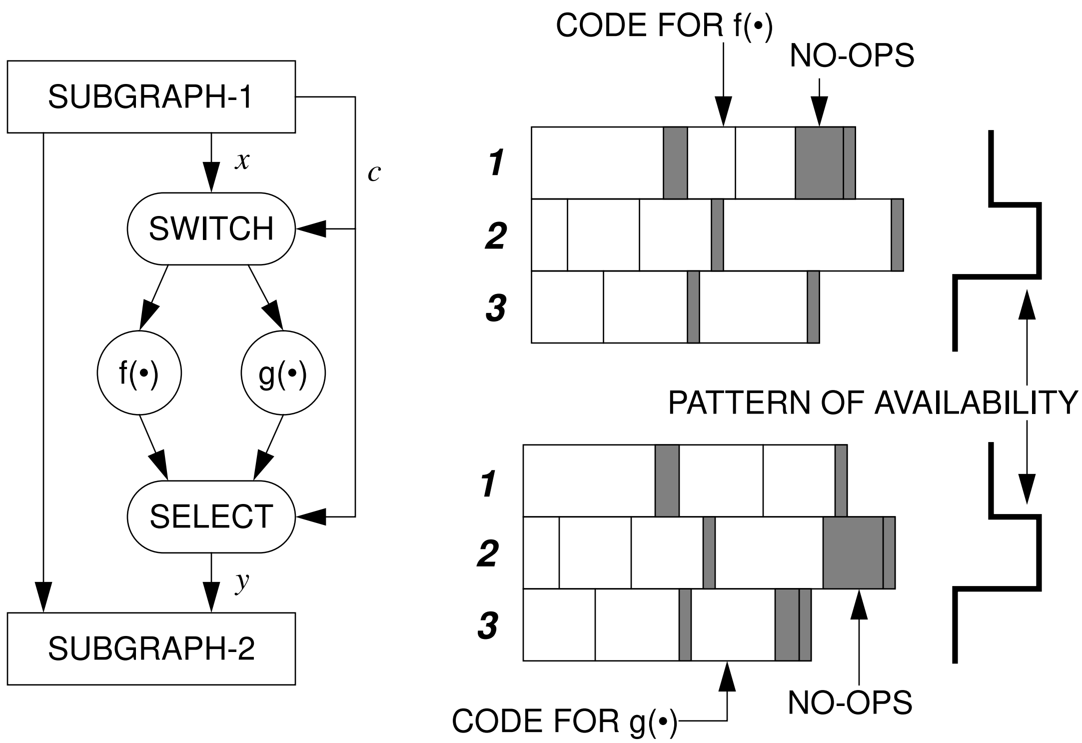


Figure 2.6 A dataflow graph containing the construct y:= if c then f(x) else g(x), where f and g represent subgraphs of arbitrary complexity. We produce Gantt charts for two schedules corresponding to two possible decisions. The schedules are padded with no-ops so that the pattern of availability after the conditional is independent of the decision [Lee88a].

ability after the execution of the dynamic construct is independent of any run-time data) can be applied to the scheduling of data-dependent iteration. The approach described in [Lee88a] accomplishes this task by devoting all the processors to the body of the iteration, but cannot exploit any parallelism between successive iterations and is wasteful if the body of the loop does not contain enough parallelism for one iteration to keep all processors busy. This flaw is addressed in [Ha91], in which the number of processors devoted to the iteration is chosen based on the probability distribution of the number of iterations (assumed known). The technique is further elaborated in [Ha92].

2.3.4 The Clock Calculus of the SIGNAL Language

While the approaches discussed so far have proven their usefulness in at least some domains, there is still a difficulty: dynamic actors are singled out for special treatment and not represented in the same way that regular actors are. Whether we choose to place dataflow graphs inside a larger control flow graph as in the internal representation of many compilers, or if we restrict the use of dynamic dataflow actors to special constructs and then apply special scheduling techniques, we are left with a two-level theory, with one approach to handle the uniform data flow and another approach to handle conditionals and iteration. At least for aesthetic reasons, it seems that a unified approach is desirable.

One such approach is to focus on the streams of data connecting the dataflow actors, rather than the actors themselves, and to associate a clock with each data stream. Rules are then defined for deriving clocks when generating one stream from another, and for determining conditions for two clocks to be considered compatible. In order to combine two streams with a pointwise operation (for example, we wish to add together two streams of integers to produce a third stream of integers) we require that their clocks be the same.[1] The rules for determining the properties of clocks are called the *clock calculus*. As a simple example, consider the following, where **x** is some stream (the language is a

1. Clocks need not be exactly the same if queuing of tokens is permitted, but this is not allowed in the SIGNAL model.

Lucid-inspired pseudocode):

```
alt := false cby not alt
x2 := x when alt
y := x + x2
```

Here the stream **`alt`** is alternately false and true (it is false followed by the inverse of itself), and therefore **`x2`** is a downsampled version of **`x`**. It is clear that the definition of **`y`** bears a strong resemblance to the inconsistent regular dataflow graph of figure 2.3, but we would have the same type of inconsistency even if **`alt`** were of unknown structure. Only if **`alt`** were always true would **`y`** be well-defined.

In SIGNAL, stream variables are considered to be signals with implicit clocks; thus we may consider that there is a time instant associated with each element of the signal. The exact values of these time elements do not appear in the analysis; only their relative ordering. Two signals defined at the same time instants are said to have the same clock. We are permitted to "observe" one signal at the time points corresponding to the clock of a different, "more frequent" signal; if we do, we will find that the "less frequent" signal is undefined at some points, which is indicated by the special symbol $\perp$. We can then define the semantics of the SIGNAL **`when`** and **`default`** operators and the effect they have on clocks. In the statement

```
a3 := a1 when a2
```

the signal **`a2`** must be a Boolean signal, and **`a3`** is a subsampling of the **`a1`** signal. **`a3`** has a value at each instant that **`a1`** has a value and simultaneously, **`a2`** has a true value. Note that **`a1`** could have a value at a point in time where **`a2`** does not have a value at all (has the "value" $\perp$). If so, **`a3`** also does not have a value at this point. Thus we could have

```
xle := x <= 0
px := x when x >= 0
y := px when xle
```

The only times when **`px`** and **`xle`** are simultaneously defined is when **`x`** is equal to zero, and in all such cases, **`xle`** is true, hence **`y`** has zero values defined only at points when **x** has zero values.

The SIGNAL default operator resembles a dataflow merge operation (except that, as always, queuing does not occur). In

```
a3 := a1 default a2
```

the signal **a3** has a value at any time instant that either **a1** or **a2** has a value. If **a1** has a value or both have a value, the corresponding **a3** value is obtained from **a1**. Otherwise if **a2** has a value, the corresponding **a3** value is obtained from **a2**.

A systematic method for determining consistency of clocks and their relations is then developed [Ben90]. We encode the state of a signal at a time instant into one of the three values –1, 1, or 0, corresponding to whether a signal is defined and false, defined and true, or undefined. We can then model relations between signals as equations on the finite field $\{-1, 0, 1\}$, using modulo-3 arithmetic. For non-Boolean signals, we treat them as Boolean signals where the truth value is unknown but it is known whether or not a signal is defined. Thus if it is known that two signals have the same clock (because one is defined in terms of the other using pointwise operators, for example) we can write

$$a_1^2 = a_2^2 \tag{2-8}$$

which is interpreted to mean that the signals **a1** and **a2** have the same clock (we will use bold face to refer to stream variables and italics to refer to the corresponding clocks). To understand this, observe that if a_1 is undefined (corresponding to a "clock code" of 0), so must a_2 be, and if a_1 is defined (corresponding to a code of 1 or –1) then a_2 must also have a code of 1 or –1.

We can also produce equations that model the relations described by the **when** and **default** operators. There are two separate cases for each, depending on whether the signals being downsampled or merged are Boolean or not. Consider

```
a3 := a1 when a2
```

assuming all signals are Boolean. We know that for the clocks, a_3 is the same as a_1 whenever a_2 is true (equal to 1), and otherwise a_3 is zero. The following equation models this:

$$a_3 = a_1(-a_2 - a_2^2) \quad (2\text{-}9)$$

It can be verified that this definition for a_3 defines the clock appropriately to match the semantics of the **when** statement (remember that arithmetic is carried out modulo 3 and reduced to the values $\{-1, 0, 1\}$). This can be verified by the truth table method, by considering all nine combinations of values for a_2 and a_3.

If a_1 is not Boolean, we only know that a_3 is defined when a_1 is defined and a_2 is true, so we have

$$a_3^2 = a_1^2(-a_2 - a_2^2)\,. \quad (2\text{-}10)$$

For

```
a3 := a1 default a2
```

we know that **a3** has a value that is the same as that of **a1** if **a1** is defined, and has a value that is the same as **a2** if **a1** is not defined. It can be verified by inspection or by the truth table method that the equation

$$a_3 = a_1 + (1 - a_1^2)\,a_2 \quad (2\text{-}11)$$

is correct for Boolean signals, and

$$a_3^2 = a_1^2 + (1 - a_1^2)\,a_2^2 \quad (2\text{-}12)$$

is correct for non-Booleans.

We can now analyze the system

```
c := x > 0; g := x when c; y := x + g
```

Again, we will use italic variables to refer to the clocks of the corresponding boldface stream variables. We have $g^2 = x^2(-c - c^2)$ and also $g^2 = x^2 = c^2 = y^2$, so we must have **c** true at every point where **x** is defined (corresponding to $-c - c^2 = 1$ whenever $x \neq 0$) in order for the clocks to be consistent.[1] In this case, the result is intuitive, but the key is to be able to reason automatically about large systems of signals.

Given a system of signal definitions in SIGNAL, each definition implies a set of relationships between signal clocks. All signals that are combined with pointwise operations have the same clock; furthermore, the **when** and **default** operators cause additional equations to be added, as we have seen. The solution to this system of equations, if it exists, results in a lattice of clock definitions in which clocks for the "less frequent" signals are subsampled versions of the clocks for "more frequent" signals in the system. We will sometimes find that there exists a particular clock, called the "master clock," such that all other clocks in the system are subsampled versions of the master clock. Systems with this property are well-defined. For other systems, there is more than one possible definition of this "master clock", and all definitions are "more frequent" than any signal in the system. Such systems are underconstrained and their execution is not determinate.

The LUSTRE clock calculus resembles that of SIGNAL in many ways, but there are some important differences that tend to make the analysis of LUSTRE systems somewhat simpler. Since there is no operator like **default** that can produce a signal that is more frequent than either input to the definition, every LUSTRE signal's clock is a subsampled version of some other signal, so that it is not possible for the most frequent clock, the "master clock", to be ill-defined. There is one additional LUSTRE operator that can produce a more frequent version of a signal, called **current**, which works like a "sample and hold" operation in signal processing. The clock of the signal

```
current x
```

is the same as that of the master clock. At the points where **x** has a value, the signal **current x** has the same value, and at other points, **current x** has the same value as the most recent value of **x**. If we consider the master clock to be one of the inputs to the **current** operator, we preserve the property that we only have clocks and the subsampled versions of the clocks, so it is relatively easy to assure that only signals with the same clock

1. This is a different result from the dataflow equivalent which requires **c** to be true always.

are combined in pointwise operations.

2.3.5 Disadvantages of the SIGNAL Approach

The SIGNAL model is a powerful and useful one. However, one disadvantage to its application is that the semantics of the language depart from dataflow in several respects. It does not naturally model queuing behavior, for example. If a system like the inconsistent model of the previous section

```
c := x > 0; g := x when c; y := x + g
```

were implemented from traditional dataflow actors (e.g. the `when` operation is implemented by a SWITCH, the `>` and `+` by homogeneous dataflow actors), we would obtain an unbounded buildup of tokens on some arcs, unless `c` is always true, but nevertheless, all the streams are defined. In SIGNAL, the definition of `y` is simply an error.

In this particular case, this is probably what is desired. However, in the more general sense in which dataflow actors are completely general and in which the only restrictions are those required by the Kahn model to assure determinism, a buildup of tokens on some arcs may be just fine (and only temporary), so that a model that permits arbitrary queueing on arcs between actors is what is actually desired. Some algorithms require this form of token buildup if there is to be any hope of implementing them; the canonical example is a parser for a context-free grammar, which requires an unbounded pushdown stack. It is not possible to implement such structures in SIGNAL or LUSTRE, precisely because of the lack of queuing. It is important in such cases to determine which arcs require unbounded memory and which do not, so that as much allocation of resources as possible can be performed at “compile time.” Given this requirement that all actors obey dataflow semantics, it appears that the SIGNAL model does not satisfy the requirement, since the actor executions are so tightly synchronized that they correspond to dataflow systems in which no more than one token is permitted on any arc. Nevertheless, the SIGNAL clock calculus has strongly influenced our work.

There is research on combining the reactive model used in SIGNAL and LUSTRE (together with its parent language ESTEREL), in which components are tightly coupled and synchronous, with a communicating sequential process approach more reminiscent of dataflow, to form a hybrid model called “communicating reactive processes” [Ber93]. The model relies on a careful separation of the synchronous and asynchronous layers, so that it is a hybrid model, not a unification.

The next chapter presents a model that extends regular dataflow directly, creating a single model that encompasses both regular dataflow actors and dynamic actors such as SWITCH and SELECT.

3

THE TOKEN FLOW MODEL

Everything should be as simple as possible, but not simpler.

— Albert Einstein

In the previous chapter, we introduced Lee and Messerschmitt's synchronous dataflow model, and demonstrated its use in compile-time scheduling of regular dataflow graphs. As this model does not support the use of dataflow actors with data-dependent execution, we examined several techniques that, in some sense, extend this model (or a related model) to support data-dependent execution while still permitting some sort of formal analysis.

We now present a model, the *token flow model,* that extends regular dataflow graphs directly, modeling actors with token flow that is not known at compile time in much the same way as regular dataflow actors are modeled. Regular (or synchronous, using the terminology of [Lee87b]) actors are simply a special case of a more general actor, which we will call a *Boolean-controlled dataflow* (BDF) actor. Conditions for graphs consisting of such actors to possess well-defined cycles, a bounded-length periodic schedule, and a schedule that requires bounded memory will be discussed.

The ideas in this chapter were first presented in [Lee91b] and further elaborated in [Buc92] and [Buc93a].

3.1. DEFINITION OF THE MODEL

A regular dataflow actor has the property that the number of tokens produced on, or consumed from each arc is fixed and known at "compile time." Boolean-controlled dataflow (BDF) actors contain the regular dataflow actors as a subset, but in addition, the number of tokens produced or consumed on an arc is permitted to be a two-valued function of the value of a *control token*. The behavior of a conditional input for an actor (an input that consumes different numbers of tokens depending on the control token) is determined by a second input for the same actor; this second input always consumes exactly one token, the control token, on each execution. The behavior of a conditional output for an actor may be determined either by an input (as for conditional inputs) or by an output; in the latter case, the output produces a single control token whose value can be used to determine the number of tokens produced by the conditional output. Given this definition for actors, the Kahn condition [Kah74] is satisfied, so that all data streams produced by the execution of BDF actors are determinate, regardless of the order in which the actors are executed (as long as constraints imposed by the availability of tokens are satisfied). Furthermore, a scheduler need consider only the number of tokens on an arc, plus the values of any tokens on control arcs, to schedule the execution of the actors, whether at compile time or run time. Because the Kahn condition assures us that all valid executions of the graph produce the same streams, we can be assured that the particular evaluation order chosen by the scheduler will not matter.

To decrease the wordiness in what follows, we will use the term *port* to describe either an input or an output of a dataflow actor, and also we will use the shorter phrase "tokens transferred by a port" instead of "tokens consumed by an input or produced by an output". Thus we can say that a control token transferred by a control port controls the number of tokens transferred by a conditional port. We use "port" rather than "arc" because a port is only one end of an arc.

3.1.1 Solving the Balance Equations for BDF Graphs

In order to extend the analysis techniques used in regular dataflow to handle BDF actors with their conditional ports, we associate symbolic expressions with conditional ports to express the dependency of the number of tokens transferred on the associated control port. In figure 3.1 we see the SWITCH and SELECT actors with their associated annotations. One possible interpretation of this figure is as follows: given a sequence of n actor executions of the SWITCH actor in which the proportion of TRUE Boolean tokens consumed by the control port is p_i, the number of tokens produced on the TRUE output of the SWITCH actor is np_i and the number of tokens produced on the FALSE output is $n(1-p_i)$. Other interpretations are possible: if the Boolean input stream can be modeled as a stochastic process, then p_i might be considered to be the probability that a randomly selected token from the input stream is TRUE (assuming that this is well-defined), in which case the annotations indicate the expected number of tokens transferred by the associated ports for a single actor execution.

Several rigorous interpretations of the p_i are possible. The most general interpretation is that the p_i are simply formal placeholders for unknown quantities that determine

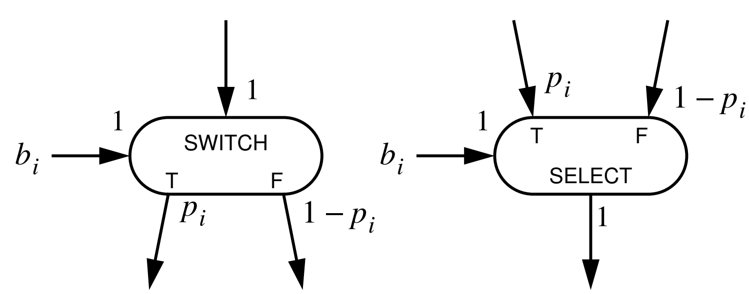


Figure 3.1 Dynamic dataflow actors annotated with the expected number of tokens produced or consumed per firing as a function of the proportion of Boolean tokens that are TRUE.

the numbers of tokens that are produced and consumed. For a probabilistic formulation, we can define p_i as the probability that a token selected from the stream b_i is TRUE provided that the Boolean stream is stationary in the mean, so that it does not matter how the sampling is performed. This condition is too restrictive for most dataflow graphs. If the stream is not stationary in the mean, but the long-term average fraction of TRUE tokens in the stream exists as a limit, this definition could be used instead, but this assumption is still too restrictive for our purposes. However, we will find that for most practical dataflow graphs, we may define p_i as the proportion of tokens that are TRUE in a well-defined sequence of actor firings, called a *complete cycle*. As it turns out, we are at no point dependent on knowing exact values for the p_i; all our manipulations will use it symbolically.

We can now use the annotated dynamic actors to analyze BDF graphs in much the same way that regular dataflow graphs were modelled in section 2.2 (and also [Lee87b]). We may combine the terms for the numbers of tokens transferred by each port into a topology matrix, and solve for the repetitions vector to determine how often the actors should be fired. As a first example, we will apply this analysis technique to the traditional if-then-else dataflow schema, shown in figure 3.2, in which we have assigned numbers to the actors and the arcs. The Boolean stream b_1 controls the SWITCH actor, and b_2 controls the SELECT actor. We obtain the following topology matrix:

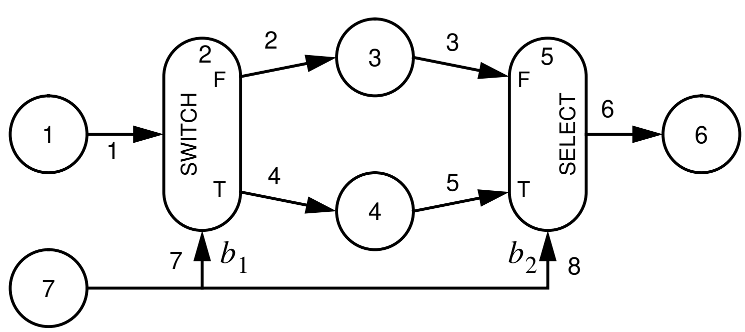


Figure 3.2 An if-then else dataflow graph. The numbers next to the arcs identify them and do not reflect the number of tokens transferred as in other figures; all actors other than SWITCH and SELECT are homogeneous.

$$\Gamma(\vec{p}) = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & (1-p_1) & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & (p_2-1) & 0 & 0 \\ 0 & p_1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -p_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \tag{3-1}$$

The topology matrix is not constant as it was for regular dataflow actors, but is instead a function of $\vec{p}$, the vector consisting of all the p variables (p_1 and p_2 in this case). We wish to find an $\vec{r}(\vec{p})$ such that

$$\Gamma(\vec{p})\,\vec{r}(\vec{p}) = \vec{o} \tag{3-2}$$

It turns out, for this example, that there are nontrivial solutions only if $p_1 = p_2$ (which fortunately is true trivially since both Boolean streams are copies of the same stream) and the solution vector has the form

$$\vec{r}(\vec{p}) = k\begin{bmatrix} 1 & 1 & (1-p_1) & p_1 & 1 & 1 & 1 \end{bmatrix}^T \tag{3-3}$$

where k is arbitrary. Note that the existence of this solution does not depend on the value of p_1. It can be interpreted to mean that, on average, for every firing of actor 1, actor 3 will fire $(1-p_1)$ times and actor 4 will fire p_1 times, which agrees with intuition. Since p_1 is not, in general, an integer, it appears to make no sense to find the smallest $\vec{r}(\vec{p})$ with integer values. Later we will see how to re-interpret repetition vectors so that the concept of a smallest integer solution again makes sense, but for now we can use $\vec{r}(\vec{p})$ to find relative firing rates.

3.1.2 Strong and Weak Consistency

Because $\Gamma(\vec{p})$ is now a function of $\vec{p}$, the existence of nontrivial solutions may also depend on $\vec{p}$. In [Lee91b], the term *strongly consistent* is introduced to describe systems such as figure 3.2 in which nontrivial solutions exist regardless of the value of $\vec{p}$.

Systems for which solutions exist only for particular values of $\vec{p}$ are called *weakly consistent*. The system we just analyzed would be weakly consistent if b_1 and b_2 were different streams, for example, because of the extra requirement that p_1 and p_2 must be equal.

Let's consider a weakly consistent system that is analogous to the synchronous language system

```
g := x when x > 0; y = x + g
```

described in section 1.4.3. As we saw, this system is inconsistent unless **x** is always greater than zero, and techniques based on the clock calculus of LUSTRE and SIGNAL can detect this. We can model an analogous system using BDF actors as well, as shown in figure 3.3. In the figure, the stream **x** is produced by actor 1 and the stream **y** is produced by actor 3 (the addition operator) and consumed by actor 4. The corresponding topology matrix is

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & p & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & 1 \end{bmatrix} \tag{3-4}$$

and we find that a nontrivial solution exists only if p is 1, as expected. The same result is obtained in SIGNAL since the stream `x > 0` is defined at exactly the same points as the stream **x**, so the complications from undefined values do not occur.

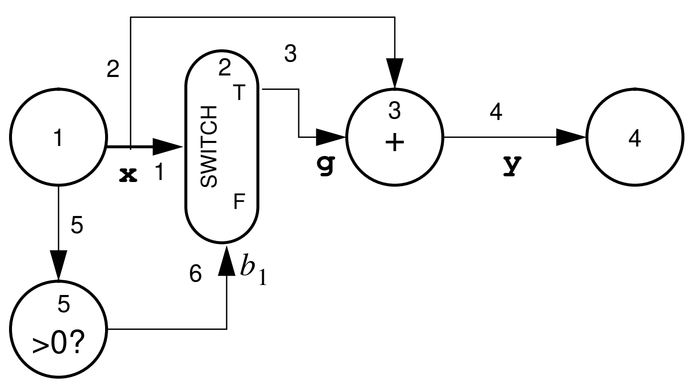


Figure 3.3 An example of a weakly consistent dataflow graph. The FALSE output of the SWITCH is not used so we ignore it.

3.1.3 Incomplete Information and Weak Consistency

When we solved the balance equations for the if-then-else graph of figure 3.2, we initially treated the Boolean control streams for the SWITCH and SELECT actors as two separate streams b_1 and b_2, and found that a condition for strong consistency was that the two streams have equal values for p_1 and p_2. In this example this is true trivially, since the two streams are identical, but it is easy to imagine cases where streams are identical but the compiler is unable to determine this, because this identity depends on mathematical properties of the actors that the compiler is unaware of or because the required analysis is too complex. In fact, since BDF graphs are Turing-equivalent, the problem of determining whether two Boolean streams in an arbitrary BDF graph are identical is undecidable.[1] As a result, a compiler that uses the techniques of sections 3.1.1 and 3.1.2 will sometimes falsely report that a BDF graph is weakly consistent, when it is in fact strongly consistent.

In most cases, the compiler will be able to report a specific reason for the weak consistency or inconsistency: in our example above, the reason might take the form "Cannot show that $p_1 = p_2$." One possibility for proceeding is to permit the user to add assertions to the graph that would explicitly provide the missing information. It would then be possible to generate code for checking such assertions at run time if desired.

While incomplete information can cause a false report of inconsistency or weak consistency, the reverse is not possible: if a BDF graph is strongly consistent, then additional information about the properties and relationships between the actors and the data streams they compute can never cause inconsistency. The effect of the additional information is, at most, a restriction of the possibilities for the vector $\vec{p}$ to a subset of $\Re^n$, where n is the number of Boolean streams. Since strong consistency implies consistency for *any* point in $\Re^n$, restriction to a subset does not alter strong consistency.

1. The Turing-equivalency of BDF graphs and related propositions are proved in section 3.4.4, assuming appropriate primitives.

3.1.4 The Limitations of Strong Consistency

If we interpret the p_i as long-term average rates, then strong consistency permits us to assert that the rates are in balance regardless of the precise proportions of Boolean tokens that are TRUE or FALSE. The analogous condition for regular dataflow graphs (that there are nontrivial solutions for the balance equations) permit us to assert that, provided that deadlock does not occur, we may compute a bounded-length schedule that executes the graph continuously in bounded memory. The fact that the schedule has bounded-length permits us to prove that a hard real-time deadline can be met, given execution times for each of the actors. For BDF graphs, however, strong consistency is not enough to assure either a bounded length schedule or bounded memory, because strongly consistent BDF graphs are easily constructed that have neither property.

Consider the modified if-then-else construct in figure 3.4. This example was discussed by Gao *et al.* [Gao92]. The only difference between this version and the one that we have seen before is that actors 3 and 4 now consume two tokens from their input arcs, and produce two tokens on their output arcs, on each execution. The result is to modify four elements in the topology matrix. The modified topology matrix is as follows:

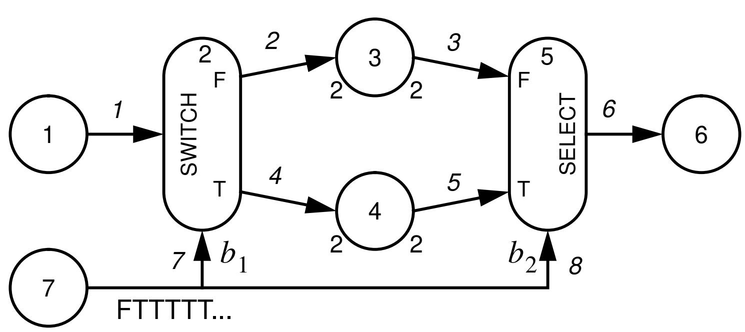


Figure 3.4 Modified if-then-else construct [Gao92]. Oblique numbers identify arcs; roman numbers next to ports indicate those inputs and outputs that transfer more than one token.

$$\Gamma(\vec{p}) = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & (1-p_1) & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & (p_2-1) & 0 & 0 \\ 0 & p_1 & 0 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & -p_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \tag{3-5}$$

Solving the modified balance equations gives

$$\vec{r}(\vec{p}) = k\,[2, 2, (1-p_1), p_1, 2, 2, 2]^T \tag{3-6}$$

Since the existence of this solution does not depend on the value of p_1, again we have a strongly consistent system. However, if all data communication on arcs is required to be FIFO, difficulties emerge. Consider, as did Gao *et al.*, what happens when actor 7 produces a single FALSE token followed by a long string of TRUE tokens, as shown in the figure. Since the control arc of the SELECT actor is FIFO, the initial FALSE token will “block up” its input. The single token on arc 2 will not be enough to fire actor 3. Actor 4 will be able to fire any number of times, but the SELECT gate will be unable to fire, since with a FALSE token on its control port it requires a token on arc 3, corresponding to its FALSE input. Whenever actor 7 produces another FALSE token, the SELECT gate will become “unblocked” because actor 3 will be able to fire. The accumulated queue of TRUE tokens will then match up with the accumulated queue of tokens on arcs 4 and 5, and execution can continue. Since the run of TRUE tokens may be of any length, either unbounded memory must be provided for or the system will deadlock.

If this system is executed on a tagged-token dataflow machine, however, unbounded memory is *not* required, since we may now execute actors out of order as soon as two tokens that are destined to be passed together to the same actor are available. In this case, we could execute the SELECT actor out of order, pairing the TRUE tokens in the queue with the data tokens on arc 5. This is permissible since there are no data dependencies between successive executions of actor 6, the sink actor. If, however, a self-loop with an initial token were added to actor 6, we would then be forced to execute it sequentially, which would again require unbounded memory.

For compile-time scheduling of BDF graphs, it would be permissible to do the same kind of rearrangement of actor executions at compile time that can be accomplished at run time by token matching. However, in the remainder of this discussion we will assume that FIFO execution is required.

3.2. ANALYSIS OF COMPLETE CYCLES OF BDF GRAPHS

We now introduce some terminology to permit us to analyze the properties of BDF graphs in more detail.

The *state* of a BDF graph consists of all information about the graph that affects the eligibility of actors for execution. For control arcs, we must know the number of tokens present, together with their values (TRUE or FALSE) and the order of their values. For other arcs, only the number of tokens is significant. Thus we might encode a state of the system in figure 3.4 as {0, 1, 0, 1, 4, 0, 0, TFFFF}. This concept is analogous to the concept of a marking for Petri nets.

A *complete cycle* of a BDF graph consists of a sequence of actor executions that returns the graph to its original state. Clearly, a null sequence of actor executions is a complete cycle under this definition, though trivial. We define a *minimal complete cycle* to be a non-null complete cycle with no non-empty subsequence that is also a complete cycle.

For any dataflow graph, we can ask the following questions:

- Do complete cycles even exist? If flow rates are inconsistent, it is possible that no sequence of actor executions will return the graph to its original state.
- Does the graph deadlock?
- Is the number of actor executions required for a complete cycle bounded, regardless of the values of any Boolean tokens produced or consumed? This condition is

useful when there is a hard real-time deadline for execution of the graph.

- Finally, can the graph be executed with bounded memory? If so, memory can be statically allocated.

3.2.1 Interpretation of the Balance Equations for BDF Graphs

For regular dataflow graphs, we determine the properties of complete cycles by solving the balance equations. Since $\Gamma\vec{r} = \vec{o}$, the result of executing actors in such a way that each actor i is executed r_i times is that the system returns to its original state. If there is only a trivial solution to the balance equations, we conclude that no minimal complete cycles exist. If the balance equations have nontrivial solutions, then either the graph deadlocks, or schedules that are bounded both in length and in memory requirements exist and are easily generated [Lee87b].

We cannot perform the corresponding analysis for BDF graphs with dynamic actors given what we have done so far since the repetition vectors are not integral given the interpretation of the p_i as probabilities or long-term averages. Accordingly, we revise our interpretation: we consider them to be the fraction of Boolean tokens on the stream b_i produced during a complete cycle that are TRUE (assuming for the time being that complete cycles exist). Since the complete cycle must restore the graph to its original state, the number of Boolean tokens of each type that are produced on a given stream is equal to the number consumed. Since tokens are discrete, this means that

$$p_i = \frac{t_i}{n_i}, \tag{3-7}$$

where n_i is the total number of control tokens produced in the stream b_i during the complete cycle, and t_i is the total number of these n_i tokens that are TRUE. We may then analyze the properties of complete cycles as follows: solve the balance equations as discussed previously when we considered the p_i to be probabilities or averages. Then substitute for the p_i using equation (3-7) above, and then constrain the number of actor executions, control tokens, and TRUE control tokens to be integral.

Let us reconsider the if-then-else and modified if-then-else examples discussed previously. When solving the balance equations for figure 3.2, we obtained the solution

$$\vec{r}(\vec{p}) = k\begin{bmatrix}1 & 1 & (1-p_1) & p_1 & 1 & 1 & 1\end{bmatrix}^T. \tag{3-8}$$

One Boolean token is produced for each execution of actor 7, thus $n_1 = k$. So, substituting equation (3-7) for p_1, dropping the subscript of 1 (since there is only one Boolean stream), and substituting n for k we have

$$\vec{r}(\vec{p}) = \begin{bmatrix}n & n & (n-t) & t & n & n & n\end{bmatrix}^T \tag{3-9}$$

We still have an integer solution if n is 1, in which case t is either zero or 1. The minimal complete cycle therefore has the repetition vector

$$\begin{bmatrix}1 & 1 & (1-t) & t & 1 & 1 & 1\end{bmatrix}^T \tag{3-10}$$

where the variable t is 1 if a TRUE token is produced by the execution of actor 7, and 0 if a FALSE token is produced. This solution is in accord with intuition.

Consider the modified if-then-else example in figure 3.4, in which the conditionally executed actors produce and consume two tokens per execution. We obtained the following solution for the repetition vector:

$$\vec{r}(\vec{p}) = k[2, 2, (1-p_1), p_1, 2, 2, 2]^T \tag{3-11}$$

Substituting using equation (3-7) for p_1 as before, noting that this time $2k = n$ where n is the number of Boolean tokens in the cycle, we obtain

$$\vec{r}(\vec{p}) = \begin{bmatrix}n & n & (\frac{n-t}{2}) & \frac{t}{2} & n & n & n\end{bmatrix}^T \tag{3-12}$$

We now seek to find the smallest integer solution for this equation. We notice two constraints for such a solution to exist: the number of TRUE tokens produced in the cycle

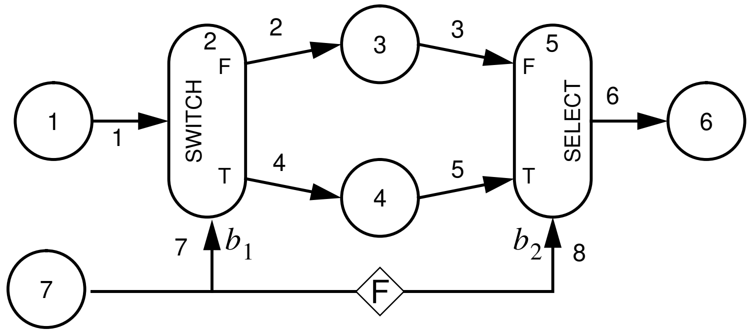


Figure 3.5 An if-then-else construct modified to have an initial FALSE token on the control arc for the SELECT actor.

must be even, and also the number of FALSE tokens produced in the cycle, $n - t$, must be even. Given these constraints, and given that we have no control over the sequence of Boolean outcomes, there is no limit to the length of the minimal cycle. In particular, if the first Boolean token is FALSE and then a large even number of TRUE tokens are produced, the cycle will not end until another FALSE token is produced.

Finally, we consider a third example, again obtained by modifying the basic if-then-else construct. In our original discussion, we treated the stream of control tokens for the SWITCH and the SELECT actors as two separate Boolean streams, and showed that the graph was strongly consistent if the corresponding quantities p_1 and p_2 are equal. We now modify the graph by adding an initial token to the control arc for the SELECT that has value FALSE, as shown in figure 3.5. Now the streams are no longer identical; b_2 is a delayed version of b_1. Initial tokens do not affect the topology matrix for a dataflow graph, as it depends only on the number of tokens produced or consumed by the actors. With a probabilistic or long-term-average interpretation, we can neglect the initial transient and still claim that this graph is strongly consistent. When computing the properties of complete cycles, however, we require that the graph be returned to its initial state (including the FALSE token on arc 8) and also that the proportion of TRUE tokens in streams b_1 and b_2 be equal. Both conditions are met if and only if the last token produced by actor 7 in the cycle has the value FALSE. By imposing this condition, we can set p_1

and p_2 equal and we obtain the same solution as for the if-then-else, with one difference: we have the extra constraint that there must be a FALSE token in the stream. Equation (3-9) is still valid. However, since every complete cycle must now contain a FALSE token, we may not reduce *n*, the number of executions of actor 7, to 1, so equation (3-10) is not valid. Instead, we have as a minimum that $n - t = 1$, and thus

$$\vec{r}(\vec{p}) = \begin{bmatrix} n & n & 1 & t & n & n & n \end{bmatrix}^T \tag{3-13}$$

where $n = 1 + t$.

As we shall see, proofs like the above that minimal cycles have unbounded length are not sufficient in themselves to prove that unbounded memory is required to execute the graph. At this point, we have merely demonstrated that unbounded time is required to return the system to its original state (assuming each actor execution requires some time). Proofs that the graphs given in figure 3.4 and figure 3.5 require unbounded memory require additional techniques and are given in section 3.4.3.

3.2.2 Conditions for Bounded Cycle Length

If a minimal complete cycle exists at all, it must satisfy the balance equations and therefore the analysis of the previous section constrains the properties of any solutions. It is possible, however, that even though bounded solutions exist for the balance equations, that no schedule, bounded or otherwise, exists that continually executes the graph, because the graph deadlocks. Therefore, to complete the proof that a graph has a bounded-length schedule, we must also demonstrate that deadlock does not occur. Formally, we have the following:

Theorem: a BDF graph has bounded cycle length if and only if two conditions hold: First, there must be a bounded integer solution to the balance equations for a complete cycle for any possible sequence of Boolean tokens produced in that cycle. Second, it must be possible, for each possible sequence of Boolean tokens produced, to construct a corresponding acyclic precedence graph (APG) for the BDF graph given the constraint that Boolean tokens with those particular values are produced, using the techniques of section 2.2.2.

In effect, we prove that the graph has bounded cycle length by construction: we first determine the exact number of times each actor is to be executed, and then determine that precedence constraints do not prevent us from executing those actors the required number of times. By specifying the exact values of the emitted Boolean tokens, we transform a BDF graph into a regular dataflow graph[1] (since, given the identity of all control tokens the flow of all tokens is completely determined), and we may then use regular dataflow graph techniques for constructing schedules.

Consider the if-then-else construct of figure 3.2. We have determined that there are two possible sequences of Boolean tokens that can be produced in a minimal complete cycle: a single TRUE token, or a single FALSE token. We can construct an APG for each of those cases, given the repetition vector from equation (3-10). These precedence graphs appear in figure 3.6.

In most cases, there is a large amount of redundancy between the precedence graphs produced given different assumptions about what Boolean tokens are produced. We therefore prefer to use a more compact structure called an annotated acyclic precedence graph (AAPG) to represent the full set of possible precedence graphs. As in the APG, each node corresponds to a single execution of an actor in the original graph; the difference is that nodes may be annotated with the condition under which they fire and arcs are labelled with the condition under which they move data. Nodes and arcs appearing in all the possible APGs have no annotations; nodes and arcs appearing in only some of the APGs (such as actors 3 and 4 in figure 3.6 and the arcs connected to them) receive annotations indicating the Boolean token values they depend on.

1. There are cases where this is not strictly true; there exist graphs that have complete cycles in which the same actor is fired twice, once with a TRUE control token and once with a FALSE control token, so that the number of tokens transferred on its arcs is not constant. It is, however, known at “compile time” so that it is still possible to construct compile-time schedules.

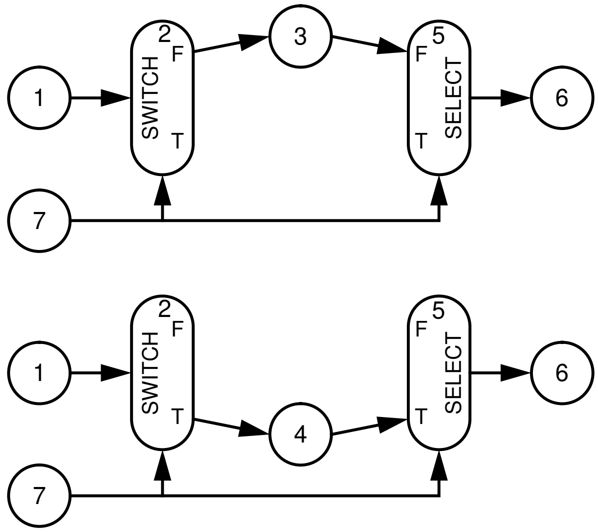


Figure 3.6 Acyclic precedence graphs for the if-then-else construct, assuming the identities of Boolean tokens produced are known. The upper graph corresponds to the production of a FALSE token, the lower graph to a TRUE token.

The AAPG can be constructed directly, rather than by combining APGs for each outcome. However, there is nothing new theoretically in this direct construction, other than bookkeeping; it is conceptually equivalent to the construction of all of the possible APGs at once.

The AAPG is a compact structure that can be taken to represent one precedence graph for each possible outcome for the generation of any Boolean tokens. As the structure corresponding to each possible outcome is bounded, we have by construction a proof that a bounded-length schedule for the graph exists. Thus successful construction of the AAPG is sufficient for a bounded-length schedule. It is also necessary, for if it is not possible to construct the AAPG then the schedule is undefined for at least some Boolean outcomes.

3.2.3 Graphs With Data-Dependent Iteration

If a graph has a bounded-length schedule, it is guaranteed that it can be scheduled to require bounded memory, because the buffer sizes return to their initial state at the end of each cycle, the number of actor firings in the cycle is bounded, and the number of tokens generated by the firing of an actor is bounded. However, the reverse is not true;

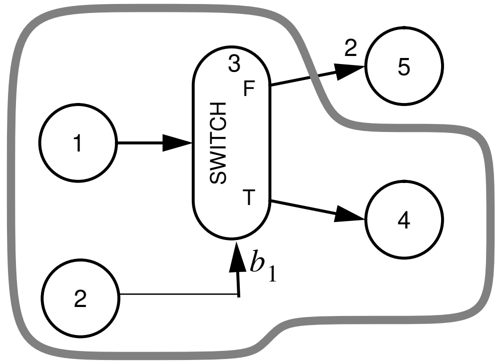


Figure 3.7 A dataflow graph that does not have a bounded-length schedule. In this graph, actors 1, 2, and 4 are homogenous, and actor 5 requires two tokens per execution. The grey curve denotes a possible clustering.

dataflow graphs that require bounded memory may nevertheless have cycles that are unbounded in length. Graphs corresponding to data-dependent iteration, where the number of times an actor is executed depends on the data itself and cannot be bounded at compile time, fall into this category.

Consider the graph in figure 3.7. It would correspond to a type of if-then-else construct except for one feature: actor 5 requires two tokens per execution. Letting n be the number of Boolean tokens per cycle and letting t be the number of TRUE tokens as before, we find by inspection that the solution vector for the graph is

$$\left[n\ n\ n\ t\ \frac{(n-t)}{2} \right] . \tag{3-14}$$

Investigating the properties of minimum integer solutions of this vector, we find that a complete cycle requires that the number of FALSE tokens generated in the cycle be even. If a TRUE token is generated first, we can immediately complete the cycle; however, if a FALSE token is generated, the cycle does not complete until we have a second FALSE token. At this point, it looks very much like the example from [Gao92] in figure 3.4. There is an important difference, however.

Consider the subsystem consisting of the actors enclosed by the grey curve in figure 3.7. Let us assume that we are given the problem of computing a separate schedule for this subsystem, excluding actor 5. Our rule for constructing schedules for disconnected subsystems is this: we will assume that any number of tokens are available from any disconnected input ports, and that we can write any number of tokens to disconnected output ports. Our desire is that the subsystem as a whole, with its internal schedule, will resemble a BDF actor from the outside[1]. Given this rule we have the following repetition vector for the subsystem: $\begin{bmatrix} 1 & 1 & 1 & t \end{bmatrix}$. The corresponding schedule might, for instance, execute actor 1, then 2, then 3, and then optionally 4 if a TRUE token was produced.

Notice that the schedule has bounded length, and therefore has bounded memory. If, however, we try to treat the cluster as a whole as a single actor, we have a difficulty: if the above schedule is executed, the cluster may or may not produce a token on its output (the input to actor 5). Consider the following solution: let us repeatedly execute the schedule until a token is produced on the FALSE output of the SWITCH actor. We have now enclosed the schedule in a do-while loop. The resulting looped schedule produces a cluster that, when executed, always emits one token; it is a homogeneous dataflow actor. We can then compute a new schedule at the top level that is also bounded in length. The resulting schedule, assuming a sequential processor, might look like the following (written in a C-like pseudocode):

```
repeat 2 times {
      do {
            actor 1;
            b = actor 2;
            actor 3;
            if (b) actor 4;
      } while (b);
};
actor 5;
```

We notice the following: if we can divide the dataflow graph into clusters in such a way

1. To actually achieve this desire (that clusters resemble a BDF actor from the outside) requires some additional conditions that will be discussed in detail in section 3.3.3.

that each cluster has a bounded-length schedule, and the top-level graph also has a bounded-length schedule, and we permit the introduction of do-while loops of the type shown here, it then follows that the graph can be scheduled in bounded memory.

3.2.4 Proof of Bounded Memory by Use of a Preamble

Another technique that may be used to prove that some graphs have a bounded-memory schedule is by use of a preamble. This technique is particularly useful for graphs with initial Boolean tokens on control arcs. In many cases, if another state is reachable from the initial state by a bounded number of actor executions, and the new state has no Boolean tokens, it is possible to show that all minimal cycles starting from the new state are bounded in length, so that the graph can be scheduled in bounded memory. Consider the graph in figure 3.8. This graph implements a do-while loop. Since there is an initial FALSE token on the control arc for the SELECT actor we know immediately that the minimal cycle length is unbounded; all cycles must end with a FALSE token on the Boolean stream produced by actor 3 to replace this token, but there is no limit to the number of consecutive TRUE tokens that may be produced. As we shall see, it is possible to apply a clustering technique to this graph, although another technique we have not yet discussed (state enumeration) is required as well. However, there is another possibility. Consider what happens if actor 1 is executed, followed by executing actor 2 (the SELECT actor).

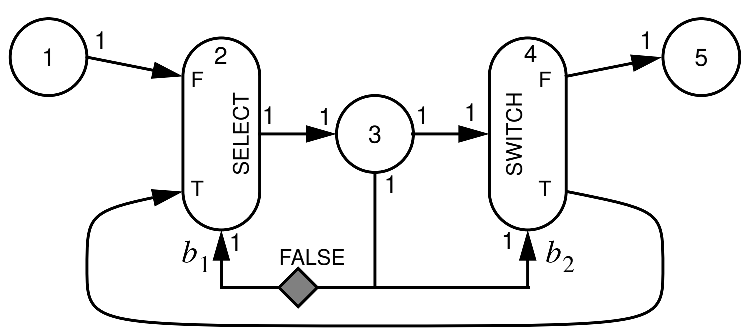


Figure 3.8 A data-dependent iteration construct corresponding to a do-while. Given an input x, actor 3 produces the data output f(x) and the Boolean output b(x). The loop repeatedly replaces x by f(x) until b(x) is false.

The resulting system is shown below in figure 3.9. There is no longer a skew between the Boolean streams b_1 and b_2, and therefore no longer a constraint that any Boolean sequence must end with a particular value. Letting n and t have their usual interpretations, we find that cycles have the repetition vector $\left[(n-t) \;\; n \;\; n \;\; n \;\; (n-t) \right]$. As there are no other constraints, for minimal cycles we have $n = 1$ and t is 0 or 1. We have bounded length cycles and therefore bounded memory. The following pseudocode represents a schedule that executes this graph "forever" using a preamble:

```
1; SELECT;
do forever {
      3; SWITCH;
      if (control token from 3 is FALSE) { 5; 1; }
      SELECT;
}
```

What is the relationship between the bounded-length cycles of figure 3.9 and the unbounded-length cycles of figure 3.8? We notice that minimal cycles for figure 3.9 contain only a single production and a single consumption of a Boolean token, while minimal cycles for figure 3.8 produce and consume any number of TRUE tokens and a single FALSE token. Therefore the relation between the two notions of cycles corresponds to the addition of a do-while loop. Repetition of a bounded-length schedule that returns the num-

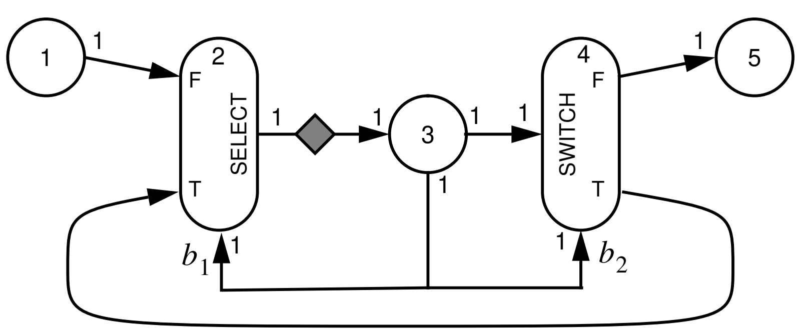


Figure 3.9 The system of figure 3.8, after executing actors 1 and 2 once each. The new system has a bounded-length schedule.

ber of tokens to the same value each time clearly keeps memory bounded, no matter how many times the schedule is repeated.

For the preamble approach described to be feasible, three conditions must hold: there must be initial Boolean tokens in the graph, and it must be possible to execute a bounded number of the actors in such a way as to eliminate these tokens (one implementation we have experimented with simulates the dynamic execution of the graph with all actors that produce Boolean tokens disabled, until either deadlock occurs or all Boolean tokens are eliminated). Finally, the resulting graph must have a bounded length schedule.

3.3. AUTOMATIC CLUSTERING OF DATAFLOW GRAPHS

As we have shown, one way to demonstrate that a BDF graph can be scheduled in bounded memory is to cluster it and show that each of the clusters has a bounded-length schedule; where necessary, subclusters are then executed repeatedly to obtain the full schedule, which then contains data-dependent iteration. In order to make this approach feasible, we require algorithms to perform the clustering.

The structure obtained by performing this clustering resembles the hierarchical "well-behaved dataflow graphs" of Gao *et al*. [Gao92]. In Gao's work, it is demonstrated that certain standard constructs corresponding to conditionals and data dependent iteration are "well-behaved" in the sense that, if the construct is treated as a cluster, it can be regarded from the outside as a single (coarse grained and composite) regular dataflow actor. A style of programming is advocated in which graphs are built up hierarchically out of these constructs. Given a graph constructed with this technique, our clustering algorithm will find the constructs, and in that sense it is precisely the reverse of Gao's approach. Given an unstructured dataflow graph, we cluster it to find structure within it. The technique is partially applicable even to graphs that cannot be scheduled with bounded memory, since even such graphs will, as a rule, contain many arcs and subgraphs that can be scheduled to use bounded memory, permitting memory to be allocated at compile time for most arcs.

3.3.1 Previous Research on Clustering of Dataflow Graphs

There have been three principal motivations for clustering of dataflow graphs. First, to improve performance on dataflow machines, it has been found useful to collect and group those actors that can be executed sequentially and treat the combined cluster as a unit; such units are sometimes referred to as *threads* because of their resemblance to communicating sequential processes (the threads can have state because of internal tokens within the cluster); the term *grains* is used in [Gra90]. The need for synchronization is thereby reduced. The compiler is responsible for rearranging and grouping the dataflow graph into clusters to accomplish this. As a rule, code for a thread is generated at compile time, and the dataflow machine dynamically selects which thread to execute depending on the availability of tokens. This approach has been used in the Epsilon-2 [Gra90] and EM-4 [Sat92] hybrid dataflow architectures, and in the Monsoon project [Tra91].

Second, clustering is used to partition dataflow graphs for scheduling on multiple processors when static assignment is used (see section 2.1). In many ways this resembles the process for collecting actors into threads for dynamic execution by a hybrid token flow machine; in either case we can consider the resulting clusters to be communicating sequential processes. A comparison of several techniques for solving this clustering problem can be found in [Ger92]; a more thorough treatment of several specific techniques appears in [Sih91] along with many references to the literature.

Finally, clustering has been used to determine the loop structure of regular dataflow graphs for the purpose of generating compact code for a single sequential processor. This work has taken place primarily in the context of research on the Gabriel [Bie90] and Ptolemy [Buc91] systems with the goal of improving code generation for programmable DSP devices. Some of this work is described in [How90] and [Bha93a]; related work with a different optimality criterion appears in [Rit93]. The problem is analyzed in considerably more detail in [Bha93b] and necessary and sufficient conditions are given for a regu-

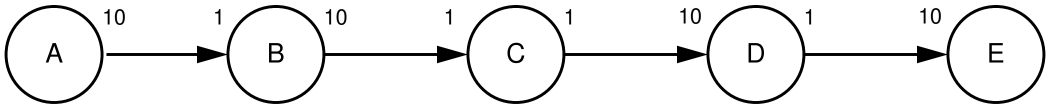


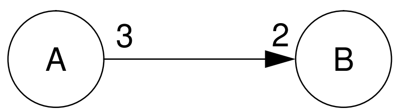
Figure 3.10 A graph that has a nicely nested single appearance schedule.

lar dataflow graph to possess a completely clustered form called a *single appearance schedule*. Single appearance schedules are defined and discussed in more detail in the next section.

3.3.2 Generating Looped Schedules for Regular Dataflow Graphs

The techniques we shall develop for clustering BDF graphs are formed by extending solutions to the corresponding problem for regular dataflow graphs. We will therefore discuss procedures for producing looped schedules for regular dataflow graphs in detail.

To motivate the problem, consider the following simple dataflow graph:



Assume that we wish to schedule this graph to execute on a single sequential processor. If our criterion is to minimize the memory needed for the data buffer between the actors, we might choose the schedule ABABB, which requires a buffer capable of storing four data tokens. An alternative that normally leads to more compact code is to choose the schedule (2A),(3B) instead, although now the buffer requires six tokens. This form of schedule, with the number of repetitions preceding each sub-schedule, is known as a looped schedule[1]; if the looped schedule contains only one appearance of each actor, it is called a *single appearance schedule*. For the graph in figure 3.10, one possible single appearance schedule is A,10(B,(10C),D),E.

A single appearance schedule (if such a schedule exists) is the goal of the looped

1. It appears that we have used the term "looped schedule" in a different sense in section 3.2.3; however, we will soon produce a unified framework that combines integer repetition factors and do-while loops into one unifying structure.

schedule generation problem. There are regular dataflow graphs that do not have single appearance schedules; they inevitably contain feedback loops of a special form called a *tightly interdependent component* in [Bha93a]. Consider, for example, the graph in figure 3.11. For this graph, which has one initial token on each arc, we must execute ABA.

We now discuss an algorithm for generating looped schedules that differs significantly from either How's [How90] or Bhattacharyya's [Bha93a],[Bha93b] algorithm. This algorithm is designed to be fast and to be generalizable to BDF graphs; there are, however, some graphs that can be looped successfully by Bhattacharyya's algorithm that are not handled successfully by this approach.[1]

For the purposes of this discussion, we say that two actors are *adjacent* if there is an arc that connects them. With respect to this arc, we call the actor that produces tokens on the arc the *source actor* and the actor that consumes tokens from the arc the *destination actor*. Two adjacent actors have the *same repetition rate* if the number of tokens the source actor produces on an arc is always equal to the number of tokens the destination actor consumes from the arc. Finally, we will call an arc a *feedforward arc* if it is not part of a directed cycle of arcs, or equivalently if there is no directed path of arcs from the destination actor to the source actor. An arc that is not a feedforward arc is called a *feedback arc*.

We will assume that the graph is connected and possesses an acyclic precedence graph (APG), implying that there are nontrivial solutions to the balance equations and that deadlock does not occur. If this is true, then we can assure that certain problematic situations do not occur — for example, we will never have a pair of adjacent actors that are

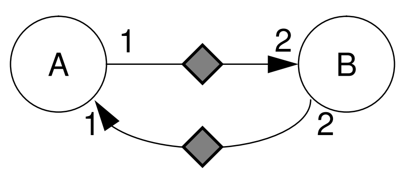


Figure 3.11 A simple graph that lacks a single appearance schedule.

1. Accordingly, the implementation in Ptolemy [Buc93], [Pin93] uses this algorithm as a first pass, applying the more general (but slower) algorithm of [Bha93a] as a second pass if the graph is not completely clustered.

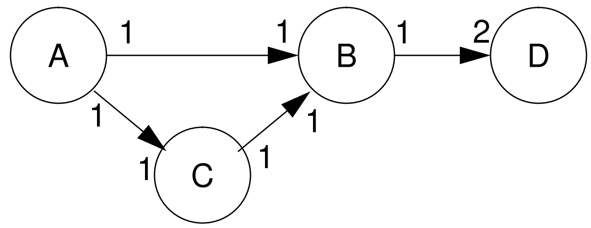


Figure 3.12 This example graph is used to help explain the loop pass.

"the same repetition rate" with respect to one arc that connects them, but not with respect to another connecting arc (this would lead to inconsistency). We would also never have arcs connecting the actors in both directions, with no initial tokens on any arc (this would be a delay-free loop and would cause a deadlock). It is possible to drop these assumptions and detect these conditions as errors with slight modifications to the algorithm; these modifications insert extra checks before a pair of actors is combined into a single cluster to test for deadlock or inconsistency.

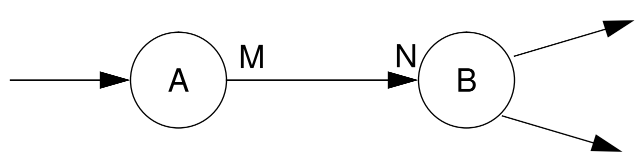
Our algorithm consists of two alternating phases: a merge pass and a loop pass. The merge pass attempts, as much as possible, to combine adjacent actors that have the same repetition rate into clusters. We must assure that no merge operation results in deadlock. In figure 3.12, for example, we cannot merge A and B into one cluster because the new cluster and actor C would then form a delay-free loop.

The loop pass may transform a cluster by adding a loop factor, corresponding to repetition of that cluster some number of times. These loop factors are chosen to cause the cluster to match the repetition rate of one or more adjacent clusters. The loop pass must also be designed to avoid deadlock, as we shall see. Loop passes and merge passes are alternated until no more transformations on the graph are possible.

The merge pass will combine an actor with an adjacent actor under the following conditions: if the actors are of the same repetition rate and are connected by an arc that has no initial tokens, the actors are always merged unless there is a directed path that starts at the source actor, passes through at least one actor that is not in the pair of candidates to be merged, and ends in the destination actor. Given the graph in figure 3.12, A and B may not be merged because of the path A, C, B. However, A and C may be merged, and the resulting cluster may be merged with B. If the only arc (or arcs) connecting the actors has one or more initial tokens, we may complete the merge given the above conditions (no indirect path) only if the connecting arc is a feedforward arc. Finally, if there are arcs of both kinds (with and without initial tokens) connecting the actors, we may ignore the presence of the arcs with initial tokens and use the arcs without initial tokens to complete the merge.

The loop pass introduces looping for the purpose of matching rates of adjacent clusters. If a loop factor of n is applied to a cluster, then each of its ports transfers n times as many tokens per cluster execution. Unrestricted looping may also introduce deadlock, for example, adding a loop factor of 2 to actor A in figure 3.11 can cause deadlock. We therefore must avoid this. It is also desirable for the generated loops to nest; in figure 3.10 we would not want to begin by looping actor B 10 times to match the rate of actor A, because we would then not wind up with A,10(B,(10C),D),E but rather something like A,(10B),(100C),(10D),E, and the latter schedule requires considerably more memory to store tokens on arcs.[1]

The loop pass has two phases: the integral loop pass (so called because it only applies to integral rate changes, corresponding to arcs where the number of tokens transferred by one neighbor evenly divides the number of tokens transferred by another neighbor) and the nonintegral loop pass (which will attempt to add loop structure to more general graphs). To understand why we separate these cases, consider the following portion of a dataflow graph, where A and B are actors or clusters:



If M evenly divides N, we could add a loop around actor A to permit a later merge

1. In [Rit93], the single appearance schedule problem is attacked with a different optimality criterion to form *minimum activation schedules*; with this criterion the latter schedule is preferred.

operation; similarly, if N evenly divides M, we could loop B. If the ratio of the smaller to the larger value is not an integer, however, we must loop both clusters, and it turns out that the conditions for making this a safe operation are considerably more restrictive.

Integral rate changes may be produced by adding a loop factor to a single actor or cluster. A cluster will not be looped if it is connected to a cluster at a different rate by an arc with initial tokens that is not a feedforward arc. To see why this rule is needed, see figure 3.11; looping actor A in that graph would introduce deadlock. Also, to make sure that the looping will nest properly, we will not loop a cluster if it is connected to a peer that "should loop first" (that is, would match the rate of this cluster if it were looped). Thus it would be forbidden to loop actor B in figure 3.10, since C should be looped first. The choice of loop factor corresponds to a choice of a peer actor for a subsequent merge; if this merge would not be permitted (because of the potential for introducing deadlock) neither would the loop be permitted.

The simple nonintegral loop pass described here is restricted to graphs that either have only two clusters or have a tree structure (only feedforward arcs). In essence, it applies a loop factor to every cluster so that all rates in the graph will match. We do not attempt to handle more complex cases here; the result is that some graphs are not completely clustered by this algorithm. Nevertheless, most common cases are handled.

3.3.3 Extension to BDF Graphs

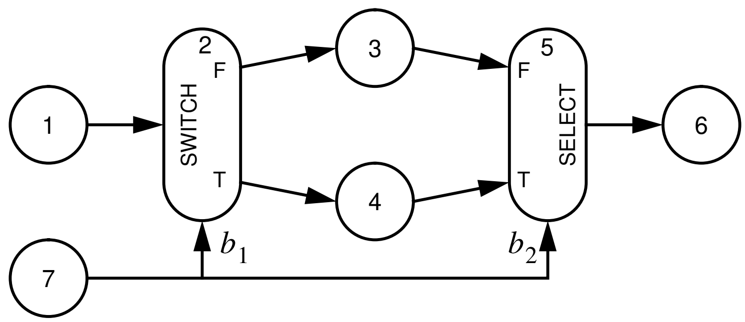
We now consider the extension of the above approach to BDF graphs. Clearly, the rules for merging the regular actors that make up the graph may proceed unchanged; doing this for these actors and leaving the resulting clusters for dynamic execution resembles the approach taken by [Tra91] to some degree. To go beyond this, we consider the meaning of adding loop factors like p_i and $\frac{1}{p_i}$ to a cluster in a BDF graph, where p_i is the rate parameter corresponding to the fraction of values in a Boolean stream that are TRUE. We shall interpret these "loop factors" as "execute this cluster only if the token from b_i is TRUE" or “repeatedly execute this cluster until a TRUE token from the stream b_i is obtained.” These interpretations are easier to understand when p_i is interpreted as t_i, the number of TRUE tokens produced or consumed on the Boolean stream, divided by n_i, the total number of tokens produced or consumed on the stream. The point is that we can treat the introduction of conditionals and of data-dependent iteration loops with Boolean termination conditions within the same framework as the introduction of iteration in regular dataflow graphs.

We are now ready to discuss the extension of the algorithm described in section 3.3.2 to BDF graphs. There are extra considerations to be taken into account: we require that each cluster produced obey BDF semantics. This means that each port of the cluster, like the ports of any BDF actor, must transfer either a fixed number of tokens or a number of tokens that depends on a token transferred on a control arc of that cluster, and that conditional input arcs be controlled by input control arcs.

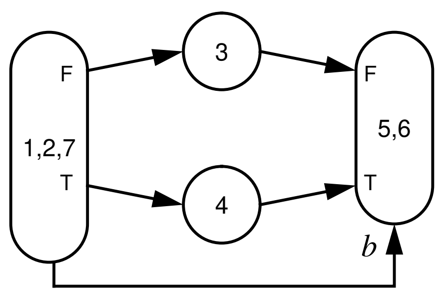
This means, for example, that we may be forbidden to merge a pair of adjacent actors because a control arc would be buried, so that the external behavior of the cluster would depend upon an invisible control signal (it is permissible to hide a control arc within a cluster as long as no arc that it controls is visible outside the cluster). We may also choose, when merging a pair of clusters connected by a Boolean control arc with initial Boolean tokens, to have the control arc appear as a self-loop in the merged cluster. In addition, we permit certain graph transformations that correspond to the combination of a merge operation and a loop operation; this sort of transformation is required when the result of the merge would bury a control arc. Finally, when the loop pass adds an “if” condition to a cluster, it is normally necessary to add an arc that passes a copy of the Boolean control stream to that cluster to preserve BDF semantics.

We will now demonstrate the above points by applying the clustering algorithm to a variety of BDF graphs. In the figures showing partially constructed graphs, we will indicate conditional ports by associating the labels “T” and “F” with them and the associated

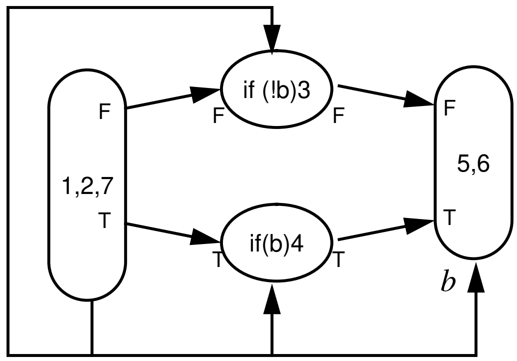
Boolean control streams by labels such as b_1 or (if there is only one stream) b. Ports with no label can be assumed to be homogeneous (such ports transfer a single token). Consider yet again the canonical if-then-else construct from figure 3.2, repeated below for convenience:



We may clearly merge actors 1 and 2; we may also merge actors 5 and 6. Exploiting the fact that all outputs of the fork have the same value, we may merge actor 7 into the cluster formed by merging actors 1 and 2 as well (we must use this fact or else the clustered graph would not contain a valid control stream for the conditional outputs of the switch). Our result now looks like this:



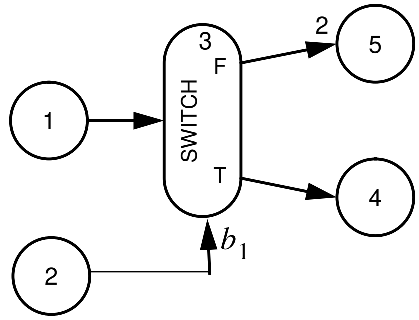
The control input for the SWITCH actor is now a control output for the cluster. Note that while the two clusters have the same rate, we cannot merge them because that would create delay-free loops involving actors 3 and 4. Therefore the first merge pass is complete. The loop pass can now convert actors 3 and 4, which unconditionally consume and produce one token, into conditional actors that match the interfaces of their neighbors. We may prepare to merge them either with the cluster containing the SWITCH or the cluster containing the SELECT. Let's suppose the former is done. For the new, conditional versions of actors 3 and 4 to be BDF actors, they require control inputs. We obtain those control inputs by adding arcs that conceptually transmit a copy of the Boolean control stream to the new actors. Our new graph looks like this:



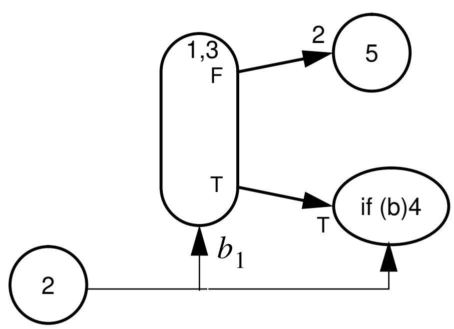
Actor 3 has now been replaced by a cluster with the following semantics: consume a control token; if it is FALSE, consume a data token, execute actor 3 using that token and output the result, otherwise do nothing. Actor 4 has been replaced by a similar conditional. At this point, all adjacent actors have matching rates so all four remaining clusters may be merged into a single cluster (at this point the control arc may be buried as it has no consequences visible outside the cluster). The clustering algorithm is complete.

It only remains to show that each cluster has a bounded-length schedule. There are three such schedules to consider (other than the trivial clusters containing only one actor): the cluster containing actors 1, 2, and 7; the cluster containing actors 5 and 6, and the top-level cluster containing four clusters. For the first two clusters, we note that no within-cluster arc has any data dependency and that all connections are at the same repetition rate; this condition suffices to assure that the schedule is bounded in length because the problem is equivalent to the scheduling of a regular dataflow graph. For the top-level cluster, some data transfers are conditional, however, the conditionals have the property that the repetition rates always match (because the algorithm was designed to assure this). As a result, we can construct a data-independent schedule for the cluster, by scheduling it as if the data transfers were unconditional rather than conditional (that is, as if all arcs labeled “T” and “F” always transferred a token). When clusters have this property, we know immediately that the graph can be scheduled in bounded memory, and furthermore, we may use regular dataflow scheduling techniques to produce code for a single processor. Conditionals arise only in the places where we deliberately added them to cause repetition rates to match.

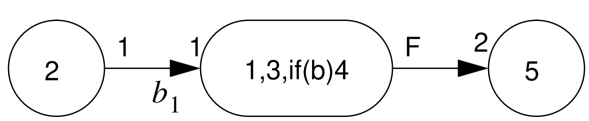
For our second example, let us consider figure 3.7, repeated below.



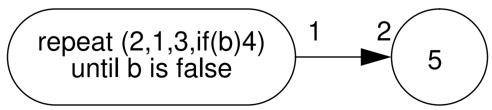
In this graph, we may merge actors 1 and 3. We are forbidden, however, from merging actor 2 with the resulting cluster, since this would "bury a control arc" — the control signal that determines which output gets data on the SWITCH actor would be hidden and we would not have a BDF actor. We can then add an "if" condition to actor 4 to cause its rate to match that of the SWITCH actor, but we cannot do the same to actor 5, since the latter actor requires two tokens per iteration. This yields the following graph:



After merging the conditionalized actor 4 with the cluster formed by actors 1 and 3, we have



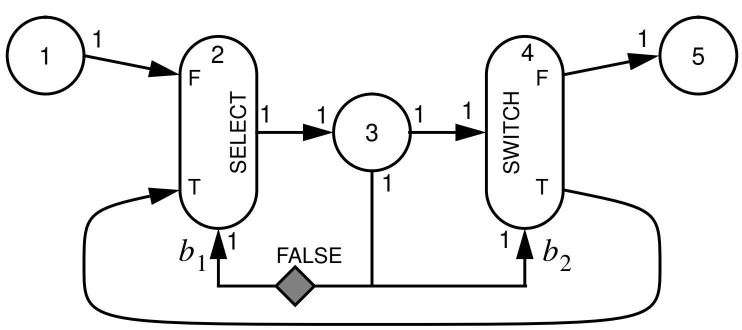
As we saw when we discussed this example earlier, we wish to introduce a do-while loop, repeatedly executing the new cluster until a FALSE token is produced. To permit this while preserving the BDF property of each cluster at each step, we must permit the merge operation (of actor 2 with the cluster it is attached to) and the loop operation (that introduces the do-while loop) to occur in one step. This operation is permissible when all outputs of the cluster would be conditional without the do-while, and would depend on a condition that appears only inside the cluster. The effect of the do-while is to make conditional ports unconditional. After the merge and loop, we now have



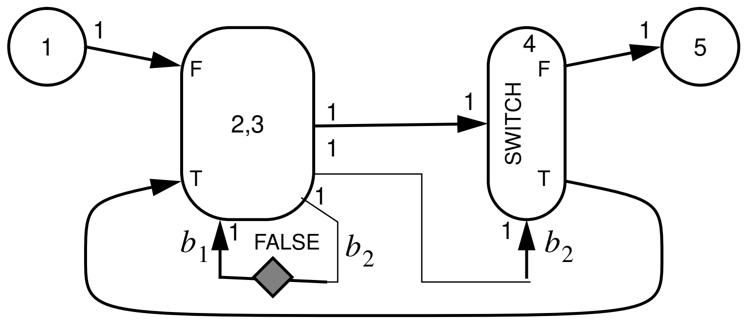
which is a regular dataflow graph at the top level; furthermore, we have a data-independent schedule at all levels. Again, the only conditional operations are those we introduced to cause the rates to match.

3.3.4 Handling Initial Boolean Tokens

When initial Boolean tokens are present, other considerations often arise. To illustrate, we will now apply the clustering algorithm to the do-while construct of figure 3.8, which we repeat below. We will not use the preamble approach, but will find a clustering that naturally reflects the control structure.

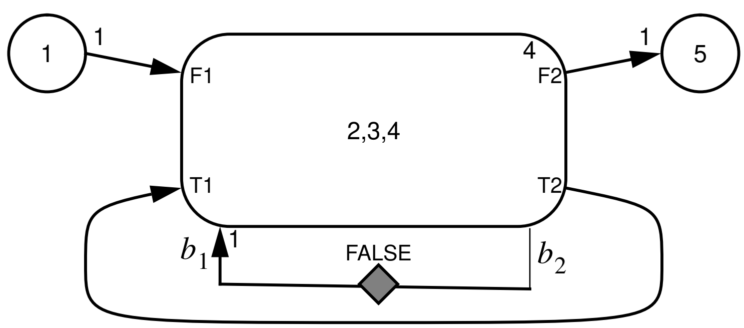


First, we merge the SELECT actor and actor 3. Because of the rule that we must keep control arcs visible, the arc with the initial delay on it becomes a self-loop of the cluster. We now have



We may now merge the SWITCH actor with the cluster we just formed, since the rates match. But there is one potential difficulty: the arc labelled "T" on the SWITCH actor is controlled by a different Boolean stream than the arc labelled "T" on the cluster (corresponding to the SELECT). We apply the following rule: for any arc with a potential rate mismatch such as this, we turn it into a self-loop rather than an internal arc when we perform the merge. This rule assures that within any cluster, all rates will match so that the cluster will always have a data-independent schedule which is bounded in length, so that only the top level of the graph retains any data dependent behavior. We can therefore always use the simpler techniques applicable to regular dataflow scheduling within clusters.

Our new system is



The labelling of conditional ports has been changed; T1 indicates that the port is controlled by Boolean stream b_1 and T2 indicates that the port is controlled by Boolean stream b_2. It would now be possible to add conditionals to actors 1 and 5 and merge them into the cluster (though it turns out that this is not desirable). If we do, however, we are left with a single actor with two external self-loops. The techniques we have developed so far do not permit us to prove that the resulting structure has a bounded-memory schedule.

For graphs with self-loops of this type, we recall that a complete cycle requires that the graph return to its original state, which includes the value of any initial Boolean control tokens. It is therefore natural to consider the following technique: consider the application of a do-while loop around the cluster with the self-loop, in which the cluster is repeatedly executed until a new Boolean token of the same type is produced. We must assure that two properties hold true: that the looped cluster possesses BDF semantics, and that the number of tokens on any self-loops remains bounded. If we apply this technique in this case, we find that the looped cluster consumes exactly one token from actor 1 and produces exactly one token for actor 5 to consume. Furthermore, by tracing execution we find that at most one token appears on the arc connecting T2 with T1.

3.4. STATE SPACE ENUMERATION

In the last example, when we verified that the introduction of the do-while loop was legitimate, we implicitly did a very simple form of state space enumeration, a process that corresponds directly to the construction of the reachability graph for Petri nets. Let us return to the previous example and treat it from a state space perspective. As it turns out, there are only two states for the cluster with the self-loop: in the initial state, there is a FALSE token on the control arc and the data feedback arc is empty. This token will be consumed and new tokens will be produced when the cluster executes. There are two possibilities: either a FALSE or a TRUE token will be produced on stream b_2. If a FALSE token is produced, no token will be produced on output T2 and the state will remain the same; otherwise, a single data token will appear on output T2. Thus there are two reach-

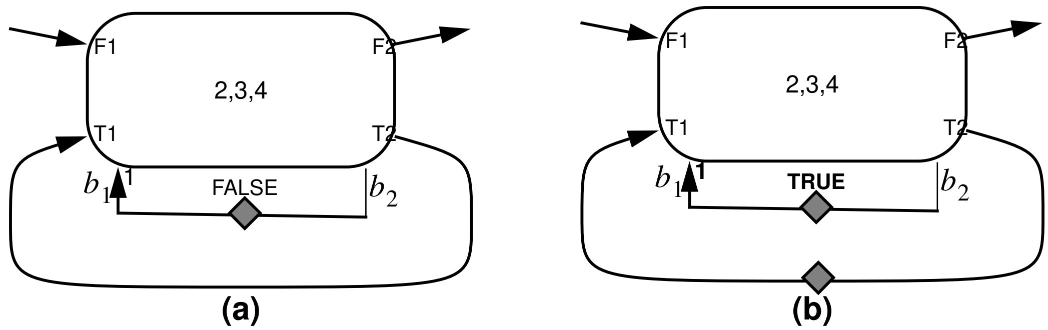


Figure 3.13 Reachable states for the data-dependent iteration cluster. State (a) is the initial state; state (b) occurs if a TRUE token is produced. From either initial state, either state is reachable as the next state.

able states, as shown in figure 3.13. Similarly, by considering the two possibilities that are reachable from the TRUE state (state (b) in figure 3.13), we find that we obtain the same two states again. We are thus assured that there is never more than one token on either visible arc. Bounds on other arcs may be obtained from the schedules for the inner clusters.

It is possible to apply a state space searching technique to the original graph, without performing any clustering. The main advantage of clustering is that the size of the state space is vastly reduced. As for Petri nets, if we can demonstrate that there are a finite number of reachable states, it follows that the memory required for arcs is bounded.

There are some significant differences between the state space search for BDF graphs and for Petri nets. First, consider the following trivial regular dataflow graph:

Interpreted as a Petri net (actors are transitions, the token storage of the arc is a place), this graph's set of reachable states is unbounded, because actor A may fire any number of times before actor B is fired. Interpreted as a regular (or BDF) dataflow graph, the graph has a schedule that is bounded in both schedule length and in memory, because we are permitted to choose the schedule AB for the graph and to avoid executing actor A a second time before the token produced from the first execution is consumed. Thus for the state space search to have a meaning, we must identify a set of rules for actor execution; these rules should be defined in such a way as to avoid ever putting more tokens on an arc than necessary.

3.4.1 The State Space Traversal Algorithm

Let us consider an algorithm that explores the state space of the graph by simulating its execution. By analogy with the reachability tree algorithm for Petri nets first given in [Kar69], we will construct a tree of reachable states. Each state will be represented by a node of the tree, with the initial state corresponding to the root node. For each node there may be multiple possibilities as to which actor to execute next, and as to the value (TRUE of FALSE) of any Boolean tokens produced by actor executions; each of these correspond to a new node of the tree that is a child of the initial state. As for the Petri net reachability tree construction, when a state that has already been reached is re-visited it will have no children. If the state space is finite, this procedure will terminate when the state space has been completely matched. If the state space is unbounded, the procedure as described so far will not terminate. We will describe a procedure to terminate the search for some such cases shortly.

To explain the rules for simulating actor execution, we require the following definitions: we define a *runnable actor* as one that has sufficient tokens on all its input arcs to execute. We say that an actor *demands input* from an arc if it requires one or more additional tokens on that arc to be able to fire. For conditional inputs, we do not say that input is demanded unless we know that a token will be required on that arc; for example, for a SELECT actor, if there is no token on the control input, the actor is not demanding input from either its TRUE data input or its FALSE data input. Finally, we define a *deferrable actor* as a runnable actor that has one or more output arcs, but no other actor demands input from any of these arcs (intuitively, an actor is deferrable if it has already produced enough data to supply the needs of all its downstream actors). For the purpose of determining whether an actor is deferrable, self loops are ignored. Actors with no output arcs other than self loops are never deferrable.

There are three possibilities at any given state that the algorithm must consider. First, it is possible that no actors are runnable at all. If so, then the graph deadlocks upon reaching this state (there are no successor states). The second possibility is that there are n runnable actors with $n > 0$, but all the runnable actors are deferrable. In this case we generate n child nodes, each obtained by executing the n^{th} runnable actor, representing the n possible next states. The final possibility is that some number $m \leq n$ of the runnable actors are not deferrable. If so, we only create child nodes corresponding to the states generated by executing each of the m non-deferrable actors. The rationale is that we never execute the deferrable actors unless the only runnable actors are deferrable. When executing an actor produces a token on a Boolean control arc, we generate two child nodes; there are two possible output states, one corresponding to the production of a TRUE token and one corresponding to the production of a FALSE token. Generation of child nodes terminates when a previously visited state is re-created.

This algorithm can generate a very large number of states. It is possible to reduce the number of states generated considerably by imposing additional constraints on the execution order (doing so is safe only if the same restrictions will apply to the scheduler). Because each Boolean token generated by an actor execution guarantees at least two successor states, one useful heuristic is to defer the execution of any actor that produces Boolean tokens as long as there are runnable, nondeferrable actors that do not produce Booleans. We may also modify the definition of a deferrable actor to specify that demands for input from actors that are themselves deferrable do not prevent an actor from being deferred; this results in a strictly demand-driven model of execution. As long as the same rules are applied in the construction of schedules or in the operation of a dynamic scheduling algorithm as are used in the construction of the reachability tree, the bounds determined by examining the nodes of the tree will be correct regardless of the details of the execution rules.

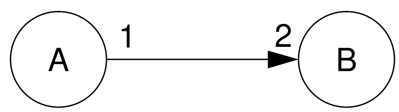
3.4.2 Proving That a BDF Graph Requires Unbounded Memory

We now consider how to cause the above algorithm to terminate on graphs for which the state space is unbounded. One simple heuristic is to terminate execution if a bound on the capacity of an arc is exceeded. The bound might be a constant for all graphs; another reasonable heuristic is to have the bound for a particular arc be some multiple of the maximum of the number of tokens written to the arc and the number of tokens consumed from the arc by its source and destination actors (the reason for this heuristic is to preserve roughly equivalent behavior as the numbers of tokens produced and consumed are scaled upward). This sort of technique is used in the Ptolemy dynamic dataflow simulator (which supports a more general model of dataflow actor than described here).

A simple bound on arc length has the weakness that it will sometimes complain about graphs that are actually bounded in memory use (because the threshold is set too low); furthermore, if the memory requirement exceeds the bound, this is not a proof that the graph is in fact unbounded. It would be desirable to have a technique that easily proves that the graphs shown in figure 3.4 and in figure 3.5 require unbounded memory. We use the reachability graph algorithm for Petri nets (as described in section 1.2.2) as a clue for how to proceed. What we require is a way to produce the equivalent of the ω places that appear in the reachability graph structure for a Petri net.

The essential feature of an unbounded Petri net that produces nodes labelled with ω in the reachability graph is the existence of a transition firing sequence that has two properties: it can be repeated indefinitely, and it results in a net increase in the number of tokens in at least one place and a net decrease in none. To apply these techniques to BDF graphs, we first require a partial ordering corresponding to the partial ordering on markings defined in section 1.2.2. We define this ordering as follows: let μ represent the state of a BDF graph. This state consists of a number (the number of tokens) for each ordinary arc and a sequence of TRUE and FALSE Boolean values for each control arc. Given two states μ and μ', we say that $\mu' \geq \mu$ if and only if the following conditions hold: for all ordinary arcs, μ' has at least as many tokens as μ, and for all control arcs, the sequence of tokens in state μ is a prefix of the corresponding sequence of tokens in state μ'. That is, given μ we can produce μ' by adding tokens of the correct type in a FIFO manner. We also define a second relation $\mu' > \mu$ that is true if and only if $\mu' \geq \mu$ and μ and μ' are distinct states.

It would now appear that we could use the procedure described for Petri nets in [Pet81] to construct the reachability tree for BDF graphs, replacing the relation used for Petri net markings with the one we have described for BDF graph states, but there is a catch. For Petri nets, any enabled actor may fire, so that given $\mu' \geq \mu$ there is no reason we could not repeat the same execution sequence that moved us from μ to μ'. For BDF graphs, however, actors that were not deferrable in the state μ may become deferrable in μ'. For example, consider the simple regular dataflow graph



Since there is only one arc and it is not a control arc, the state of the graph is a scalar and there are three states, corresponding to 0, 1, or 2 tokens on the arc. Using the number of tokens as the state name, state 1 is reachable from state 0, and state 2 is reachable from state 1. By analogy to the Petri net reachability graph construction, we might argue that we could repeat the sequence of actor executions (execute actor A) that got us from state 0 to state 1 indefinitely and therefore this graph is unbounded. This is prevented by the rule for deferrable actors, however. Since actor A becomes deferrable in state 2, it is not possible to produce more than two tokens on the arc, and the system only has three distinct states.

We therefore define a new operator on states, which returns a vector with an integer value for each arc. The value for an arc represents the number of tokens demanded on that arc, using the criterion discussed earlier: the number of tokens that must be added to satisfy the requirements of the actor that consumes from the arc. If these requirements are unknown because the arc is conditional and there are no tokens on the corresponding control arc, the number demanded is zero. We write this operator as $D(\mu)$ and refer to it as the demand vector for state μ.

If $D(\mu') = D(\mu)$, then the set of runnable actors in state μ and the set of runnable actors in state μ', as well as the set of deferrable actors, is exactly the same. This is because the demand vector completely determines this information. What we require is a sufficient condition for showing that we can indefinitely repeat the firing sequence that moves us from state μ to state μ'. The following conditions are sufficient:

- μ' must be reachable from μ,
- $\mu' > \mu$,
- $D(\mu') = D(\mu)$ (the demand vectors in both states are the same),
- an additional requirement on intermediate states between μ' and μ must be satisfied.

The fourth condition is as follows: consider all the intermediate states between (but not including) μ and μ' (on any path). Let us name these states s_i, $i = 1 \ldots n$. If, starting at state μ', we repeat the same actor executions (and assume the same results for any generated Boolean tokens) we obtain new states s'_i, $i = 1 \ldots n$. If, for each i, we have $s'_i > s_i$ and also $D(s'_i) = D(s_i)$, it follows that we can repeat the execution sequence endlessly and therefore all arcs that increase in length between the two states are unbounded. Note that if there are no intermediate states, because state μ' is directly reachable from state μ, then the first three conditions are sufficient.

Given these conditions, we can now define the state reachability tree construction algorithm as follows, using terminology borrowed from [Pet81]. We will use the ω label to indicate an ordinary arc with an unbounded number of tokens, corresponding to [Pet81]; we require a new notation for unbounded sequences. Because we must be able to compute the partial order relationship, we will represent unbounded sequences by a prefix, followed by a sequence of tokens that may be repeated an indefinite number of times, followed by an asterisk. For example, a state might be labelled as {ω,0,F(T)*}, indicating that the control arc has a single FALSE token followed by an indefinitely long sequence of TRUE tokens.

Let a *frontier node* refer to a node that has not yet been processed by the algorithm. Initially, the tree has one frontier node, the root, corresponding to the initial state. For each node, we record the number of tokens stored on each ordinary arc and the sequence of Boolean tokens on each control arc. We also store $D(\mu)$, the demand vector for the state. The processing is as follows:

If there exists another node y in the tree that has the same marking as the current node x, we stop; x is a duplicate node. If there are no runnable actors in state x, we stop; x is a deadlock state (a terminal node in the terminology of [Pet81]). Otherwise there will be successor states, which will correspond to child nodes in the tree.

We now compute all of the successor states and add a child node for each, following the rules described in section 3.4.1 for determining which actors to run. Consider a particular state and a particular actor to be fired, with particular Boolean outcomes. If the number or sequence corresponding to an arc in state x does not have an ω symbol or an asterisk corresponding to an indefinite number of tokens, the appropriate number of tokens is simply added or removed. If an ordinary arc has an ω symbol, the corresponding arc in the successor state also has an ω symbol. If Boolean tokens are added to an arc that has an indefinitely repeated sequence, the added tokens are ignored (we pretend that there are so many tokens that the “tail end” is never reached). If the actor execution consumes tokens from the beginning of a Boolean arc with a repeated sequence, we represent this in the next state by removing the appropriate number of tokens from the stream.

Finally, if for any of the newly created states μ', we can find another state μ on the path from the root such that $\mu' > \mu$, $D(\mu') = D(\mu)$, and the corresponding relations for intermediate states hold as described on page 98, we replace the new state as follows: all non-control arcs that gained tokens get a ω in their representation, and for control arcs, the marking is represented by putting the sequence of tokens added by going from μ to μ' in parentheses and adding the asterisk. For example, if we go from F to FT, we change the FT to F(T)*.

Just as does the reachability tree for a Petri net, this BDF reachability tree structure we have defined loses information, as it does not represent the reachability set (to use the terminology of [Pet81]). In states with multiple ω labels we discard any relationship between them, and we also discard suffixes added to states with infinite Boolean sequences. If there are no nodes with a ω or indefinitely repeated Boolean sequence, however, the reachability tree specifies the state space and allowed transitions completely and the BDF graph is bounded.

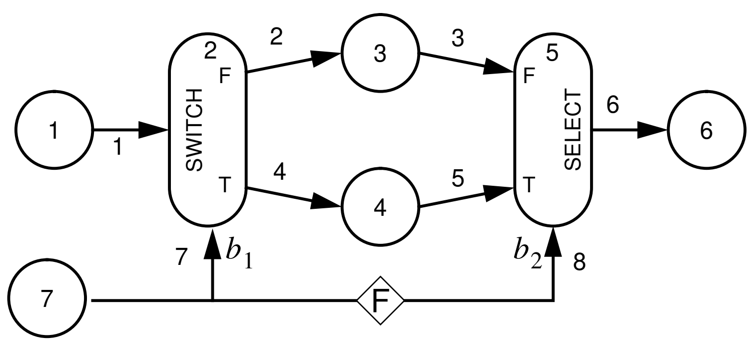
For Petri nets, it is proved in [Kar69] and in [Hac74] that the reachability tree is always bounded in size, so that the construction algorithm is a true algorithm. Is the same true for the BDF reachability tree? Unfortunately, no. The essence of the proof in [Hac74] is to show that no infinite path of nodes starting at the root can exist. In essence, what is shown is that any such path must contain an infinite non-decreasing subsequence of states such that $\mu_0 \le \mu_1 \le \mu_2 \ldots$. Since for each pair of states such that $\mu \le \mu'$, the algorithm replaces at least one place in μ' by ω, and since the number of places is bounded, we quickly reach a marking where every element is ω, meaning that the infinite non-decreasing sequence must have repeated states, which is not permitted. However, since for control arcs we record the sequence of tokens and not just their number, we can have infinite sequences of states in which the number of tokens on a Boolean arc continually increases, but in which no state's marking is a prefix of any other state's marking. For example, F, TF, TTF,... is such a sequence. The consequence is that the techniques used in [Hac74] cannot be used to prove that the reachability tree construction is bounded. This means that we may still require heuristics such as a bound on arc capacity to make the state traversal algorithm terminate.

As with Petri nets, we can convert the BDF reachability tree into a reachability graph by replacing duplicate frontier nodes with arrows pointing to the previously generated copy of the node.

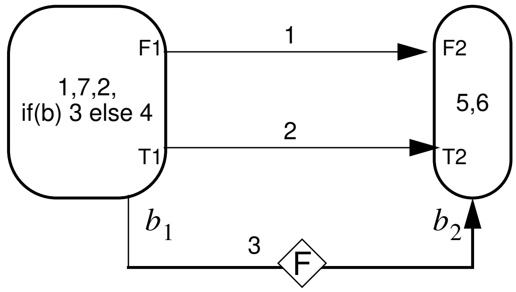
3.4.3 Combining Clustering and State Space Traversal

Clustering and state space traversal are best applied in combination. Graphs corresponding to the dataflow schema of Dennis [Den75a] or Gao [Gao92] are clustered readily; the only state space traversal needed is the simple two-state space corresponding to the node with the self-loop (see the beginning of section 3.4 for a discussion), and this is easily handled as a special case and does not require the full algorithm we have described. More irregular dataflow graphs, or graphs that do in fact require unbounded memory, may only be partially clusterable.

Let us again consider the graph in figure 3.5, repeated for convenience below:

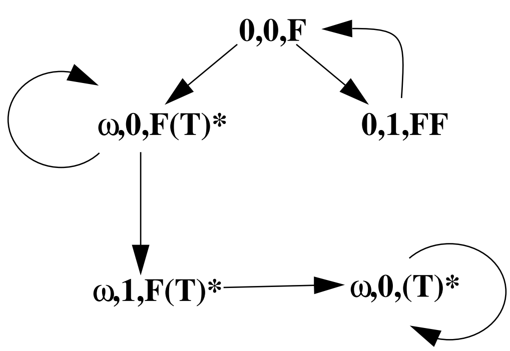


If we apply the clustering algorithm, this graph is reduced to the following structure (where as before, the label T1 on a port indicates that a token is transferred only if a TRUE token appears in the corresponding position on stream b_1):



This choice of clusters is not unique, by the way; we have chosen to combine actors 3 and 4 with the cluster containing the SWITCH actor, but we could equally well have grouped them with the SELECT actor, resulting in the same top-level pattern of clusters but with different cluster contents.

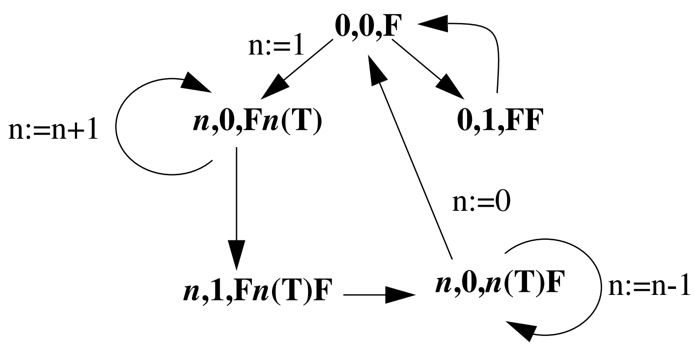
Since we can cluster no further, we now apply the state enumeration algorithm to this graph. The initial state is {0,0,F}, which we will call μ_0. There are two possible successor states: μ_1, corresponding to {0,1,FT}, and μ_2, corresponding to {1,0,FF}. We note that $\mu_1 > \mu_0$ and also $D(\mu_1) = D(\mu_0) = \{1, 0, 0\}$ (the sink cluster demands a token on its F2 input). Since there are no intermediate states, we have all that we need: arcs 1 and 3 are unbounded, and the transition that makes the arcs grow indefinitely corresponds to the production of a TRUE output by the first cluster. The complete reachability graph for this figure is



and we see that arc 2 is bounded (never has more than a single token).

The reachability graph omits some information, just as does the corresponding structure for a Petri net. Given two arcs with ω values, for example, the graph does not specify any relationship between them (though they might always have the same number of tokens). Also, given a Boolean arc with a description like (T)*, we pretend that the effect of adding tokens, whether TRUE or FALSE, does not change the description of the arc. Loosely, there are so many T's that we will never reach the end to see what is beyond. In many cases this substantially reduces the size of the graph.

However, it is possible to use a similar notation to record the entire set of reachable states if that is what we require. What is missing in the above is that the number of tokens on the first arc equals the number of TRUE tokens on the third arc in all of the states; also, suffixes are dropped in the Boolean sequence on some nodes of the graph. The following figure represents the complete state space of the graph:



In this figure, arcs labelled with assignments to the variable n reflect the value that variable has when the arc is traversed. It is clear from this diagram that the network is "live" in the sense that every state is reachable from every other state, something that is not clear from the previous figure.

The example given in figure 3.4 can be proven unbounded in a similar way. In this case, the result of clustering is the graph shown in figure 3.14.

This system has a considerably larger number of states. However, since we know that sequences such as FTTTT... are the troublesome ones, we can use this fact to quickly construct a proof of unboundedness. We find that there are states {2,0,1,0,FTT} and {3,0,1,0,FTTT} that satisfy the conditions for a proof of unboundedness: the second is reachable from the first in a single step, the demand vectors are the same and the partial order relationship holds. We therefore know that arcs 1 and 5 are unbounded. Similarly, by reversing TRUE and FALSE we find the states {1,0,2,0,TFF} and {1,0,3,0,TFFF} with the same properties, so that arc 3 is also unbounded. Note that our definition of "deferrable" prevents actors 3 and 4 from being executed until their output is demanded, therefore only two tokens are needed on their output arcs, arcs 2 and 4. In fact, if a demand-driven evaluation technique is used, any arc whose source actor has only one output is always bounded, since the source actor will not be executed if the number of tokens on the arc is sufficient to satisfy the demand of the destination actor.

It is not necessarily an error for an algorithm that is represented as a dataflow graph to require unbounded memory. As a simple example, a recognizer for a context-free grammar requires an unbounded pushdown stack. But even systems that require unbounded memory normally require this memory for a small subset of the arcs that make up the entire graph. The combination of clustering and state traversal discussed here permits such arcs to be isolated, so that a code generation model need supply dynamic memory allocation for tokens only where needed. For example, in the above figure actors 3 and 4 might represent arbitrary clusters with internal bounded-memory schedules.

3.4.4 Undecidability of the Bounded Memory Problem for BDF Graphs

We have supplied techniques for determining that some BDF graphs can be scheduled with bounded memory, and techniques for showing that others require unbounded

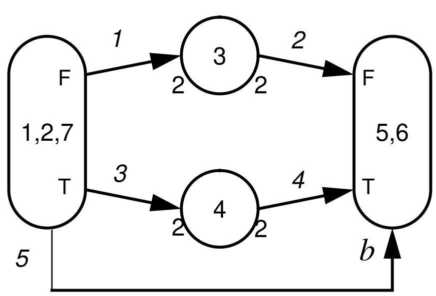


Figure 3.14 Result of applying the clustering algorithm to figure 3.4. As before, italic numbers identify arcs; non-italic numbers adjacent to inputs and outputs give the number of tokens transferred by that port.

memory. However, there are also graphs that fall “between the cracks,” not responding to any of the techniques described so far. Is it conceivable that further research will provide a complete decision procedure? The answer is no, as we shall show, using the following reasoning:

- A small set of BDF actors has equivalent computational capability to a universal Turing machine, in fact, a universal Turing machine (UTM) can be built from this small set of actors.
- If a decision procedure exists for determining whether a BDF graph has a bounded-memory schedule, it would then be possible to determine whether a Turing machine accesses a bounded or unbounded length on its tape. The latter problem is undecidable (equivalent to the halting problem).
- As a simpler alternative to building a UTM, it is possible to demonstrate the Turing equivalence of the BDF model using partial recursive function theory.

We now provide an outline for the construction of a two-tape universal Turing machine from BDF actors.[1] The building block for the data tape is a stack with the property that, if “popped” when empty, a “fill symbol” (corresponding to the blank tape symbol of the UTM) is returned. One such stack represents the tape to the right of the “head” of the UTM, and another represents the tape to the left of the “head.” The tape head can be shifted in one direction or the other by popping a token from one stack and pushing it onto the other stack. To implement a stack using BDF, we have a bit of a problem: dataflow arcs work like queues, not stacks. If we “push” onto the stack by adding a token to a queue, to “pop” the stack it is required to circulate the entire queue around and extract the last token. This is most easily accomplished if an integer-valued token is kept that gives the count of tokens on the stack.

1. We will not give the full construction, which is about as interesting as the result of the traditional assignment that a student build a computer out of NAND gates, but will just present enough to demonstrate the main design problems and show that it can indeed be done.

The program for the UTM consists of a set of quintuples: current state, current tape symbol, new state, new tape symbol, and action (e.g. shift left, shift right, halt). These reside on a set of five self-loop arcs. To determine the action, the controller block reads the current state and tape symbol, circulates the "program" around until a match is found, and generates the next state, tape symbol, and action.

To implement the UTM, we require the SWITCH and SELECT actors, together with actors for performing addition, subtraction, and comparison on the integers, plus a source actor that produces constant stream of integer-valued tokens and a fork actor.

It is perhaps easier to show that BDF graphs (using the same simple set of actors described above) suffice to compute any partial recursive function. To define the set of partial recursive functions, we first define a smaller set of functions, the set of primitive recursive functions. This set of recursively generated functions is defined to include the following functions on the nonnegative integers [Boo89]:

- The zero function, $z(x) = 0$.
- The successor function, $s(x) = x + 1$.
- For any integers M and N such that $M \le N$, the identity function of N arguments, which returns the M^{th} argument: $id_M^N(x_1, \ldots, x_N) = x_M$.
- Any function that can be expressed in terms of other primitive recursive functions using function composition.
- Any function that can be defined in terms of two other primitive recursive functions f and g using the operation of *primitive recursion*, which is defined as follows:

$$h(x_1, \ldots, x_{N-1}, 0) = f(x_1, \ldots, x_{N-1}) \tag{3-15}$$

$$h(x_1, \ldots, x_{N-1}, s(x_N)) = g(x_1, \ldots, x_N, h(x_1, \ldots, x_N)) \tag{3-16}$$

This operation defines functions by mathematical induction on the last argument. It is easy to see that addition can be defined using id_1^1 for f and the composition of s and id_3^3 for g. Similarly, by applying primitive recursion we may obtain multiplication from addition and exponentiation from multiplication.

The set of primitive recursive functions, together with the operation of minimization

$$f(x) = \{\text{The least value of } y \text{ such that } g(x, y) = 0\} \qquad (3\text{-}17)$$

(where x and y are integers), as well as composition and primitive recursion over previously defined functions, generate the set of all partial recursive functions.

Any computational procedure that computes all such functions is Turing equivalent. In order to compute all partial recursive functions, it suffices (as is shown in [Den78]) to be able to support arithmetic on arbitrarily large nonnegative integers together with a loop construct controlled by a predicate (such as “less than”). The small set of BDF actors described earlier in this section suffices to do this, therefore the BDF model is Turing equivalent.

Theorem: *the problem of deciding whether a BDF graph can be scheduled with bounded memory is undecidable*. To show that Turing equivalence of the BDF model implies that the bounded memory decision problem is undecidable, it is sufficient to show that given a bounded memory decision algorithm, we could then solve the halting problem. Assume we have an algorithm A that can determine whether a UTM uses only a bounded length of its tape with a given program and input. If we apply algorithm A and find that an unbounded length of tape is used, we know that the program does not halt. If a bounded length of tape is used and that bound is less than or equal to N, we know that the system has no more than S^N states, where S is the number of distinct state symbols. We execute the system this number of times and see if there is a loop (a repeated state). If there is, we know the system will not halt; otherwise it must have halted (since all possible states have been visited). Since algorithm A solves the halting problem but the halting problem is undecidable, it follows that algorithm A does not exist.

Theorem: *it is not possible in general to prove that two Boolean streams in a BDF graph have identical values, thus the problem of determining that a BDF graph is strongly consistent is undecidable*. To demonstrate this, we assume we have a procedure that determines that two Boolean streams are identically valued, and consider a UTM constructed out of BDF actors. We now construct a Boolean stream whose n^{th} value is TRUE if the UTM has not halted after n steps and is FALSE otherwise. We construct a second Boolean stream that is always FALSE. If we had a decision procedure that could tell whether these two streams were identical, we would have a tool for solving the halting problem, which is impossible.

In the discussions above we have used, in addition to Boolean tokens, arbitrarily large integer-valued tokens. The state traversal algorithm we have described discards information on arcs with token values that are not Boolean. However, we could equally well construct BDF graphs in which FALSE is treated as the Turing-machine “blank token”, TRUE is treated as the Turing-machine “tally” token, and the integer n is represented as $n + 1$ consecutive TRUE tokens. All arcs would then have Boolean tokens and the state as represented in the algorithm of section 3.4.3 would represent all the information about the system. It therefore follows that the state traversal algorithm does not terminate for at least some graphs (without a heuristic to cut off search).

3.5. SUMMARY

This chapter has presented a variety of techniques for the analysis of BDF graphs. Each is by necessity only partially applicable, owing to the Turing-completeness of the model which implies that many analysis questions are undecidable. However, by applying the techniques, we may divide the set of all BDF graphs into three categories.

The first category includes those graphs with bounded-length schedules. This category includes the set of all regular dataflow graphs, and it also includes constructs of the if-then-else form. The fact that the schedule is of bounded length may (depending on the semantics of execution of a minimal complete cycle) permit us to establish that hard real-time deadlines are successfully met, given execution times for each actor. Parallel scheduling techniques that apply to regular dataflow graphs are not difficult to extend to this type of graph, particularly if a minimax scheduling criterion is applied (make the worst case run as rapidly as possible).

The second category, a superset of the first, includes all graphs that may be proven to have bounded memory by clustering and state enumeration. Such graphs may express data-dependent iteration as well as conditional execution. Because of the undecidability of the bounded-memory problem, the boundary of this category is not computable and depends on the particular clustering technique used; there is still considerable room for improvement in BDF clustering algorithms.

The third category of BDF graphs are those that are not completely clusterable, and either we can prove that unbounded memory is required or we are unable to prove that the state enumeration algorithm will complete without a heuristic bound. For such graphs, it is possible to construct static schedules for the clusters, but dynamic scheduling of clusters, plus some degree of dynamic memory allocation, is needed to execute such graphs.

4

IMPLEMENTATION IN PTOLEMY

I would rather write programs that write programs than write programs.

— Anon. (graffiti from Stanford CS department quoted in [Flo79])

This chapter discusses implementation of Boolean-controlled dataflow graph analysis, clustering, scheduling, and code generation using the algorithms described in the previous chapter together with earlier work described in [Pin93]. The Ptolemy framework for heterogeneous simulation and software prototyping was used [Buc93]. We will first discuss the relevant features of the Ptolemy system in detail and then describe the features of the BDF implementation.

4.1. PTOLEMY

Ptolemy is an environment for simulation, prototyping, and software synthesis for heterogeneous systems. It uses modern object-oriented software technology to model each subsystem in a natural and efficient manner, and to integrate these subsystems into a whole. The objectives of Ptolemy encompass practically all aspects of designing signal processing and communications systems, ranging from algorithms and communication strategies, through simulation, hardware and software design, parallel computing, and generation of real-time prototypes.

Ptolemy is the third in a series of design environment tools developed at the University of California, Berkeley; its ancestors are Blosim [Mes84] and Gabriel [Bie90]. Blosim's primary focus was on algorithm development for digital signal processing; it used a general dynamic dataflow model of computation. Gabriel was designed to support real-time prototyping on parallel processors, and in addition to its use as a simulation tool, was capable of code generation for one or for multiple programmable digital signal processors. Gabriel's code generation abilities could be used only for algorithms with deterministic control flow that could be described by regular dataflow graphs. This restriction permitted the development of several automated scheduling and code generation schemes [Bha91][Lee87a][Sih91].

Unlike its predecessors, Ptolemy is not restricted to a single underlying model of computation. Instead, as a heterogenous system, Ptolemy is designed to support many different computational models and to permit them to be interfaced cleanly. For example, a Ptolemy simulation may contain a portion that uses a discrete-event model, another portion that uses a regular dataflow model, and a third portion that uses a gate-level logic simulation model. Some parts of the application might be simulated within the workstation running the Ptolemy process, while other parts might consist of synthesized DSP code running on an attached processor.

Ptolemy relies heavily on the principles of object-oriented programming to permit distinct computational models to be seamlessly integrated. In [Boo91], Booch defines object-oriented programming as follows:

> *Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of*

a hierarchy of classes united via inheritance relationships.

In Ptolemy, different computational models can be seamlessly integrated because the objects that implement them are inherited from common base classes and therefore provide the same interface, while the derived classes implement the specific behavior required for implementing specific computational models.

While it is not fundamental to Ptolemy, the graphical user interface deals with descriptions of systems represented as block diagrams (a text interface is also available). It is therefore convenient to think of the basic module in Ptolemy as a *block*, and in fact all actors in Ptolemy are members of classes derived from the class **`Block`**. An atomic block is called a *star* (and is, in fact, an instance of a class derived from the class **`Star`**). The class **`Galaxy`** represents a hierarchical block (a block that contains other blocks). The outermost block, which contains the entire application together with means for controlling its execution, is an instance of the class **`Universe`**. The entity that controls the order of execution of the blocks is the *scheduler*; some schedulers determine the entire order of execution of blocks at compile time; others do some of the work at compile time and some of the work at run time. Another important class is **`Target`**; target objects model or specify the behavior of the target of execution for code generation applications and may also provide parameters that control a simulation. The combination of a scheduler, a set of blocks, and other support classes that conform to a particular model of computation is called a *domain*. Different models of computation (time-driven, event-driven, etc.) can be built on top of Ptolemy by simply substituting different domains. Two or more simulation environments built on top of Ptolemy may be combined into a single environment, thus enabling the user to perform heterogeneous simulations of large systems that combine different computational models.

New domains are easily added to Ptolemy, including domains that do not conform to the block/scheduler model described above. In addition, new blocks and domains may be added to a running Ptolemy system by means of incremental linking. The basic interfaces that glue the system together form the Ptolemy kernel, described in detail in [Buc93c]. While Ptolemy was first conceived of for simulations, it also subsumes and extends the multiprocessor code generation capabilities of Gabriel. When these capabilities are added to Ptolemy's multi-paradigm simulation capabilities, a powerful platform for hardware-software co-design results [Kal92][Pin93].

4.1.1 Example of a Mixed-Domain Simulation

Consider the system in figure 4.1. This is a simulation model in which compressed speech is transported on a broadband packet network. The simulation requires a combination of signal processing (including compression, silence detection, and reconstruction) and queueing (packet assembly, disassembly, and transport).

The simulation naturally divides into two pieces, the signal processing (the compression, silence detection, and decompression) which is naturally modeled with a time-driven synchronous model (the SDF domain, corresponding to regular dataflow), and the network simulation (packet assembly, switching, queuing, and disassembly), where a model that only takes into account changes in system state is appropriate. Here the DE (discrete-event) domain may be used. Since neither model is preferred by Ptolemy, it is possible to design the simulation with either domain at the top level.

To connect the two domains together, a concept known as a *wormhole* is introduced. A wormhole (from astronomy and cosmology) is a theoretical object that connects

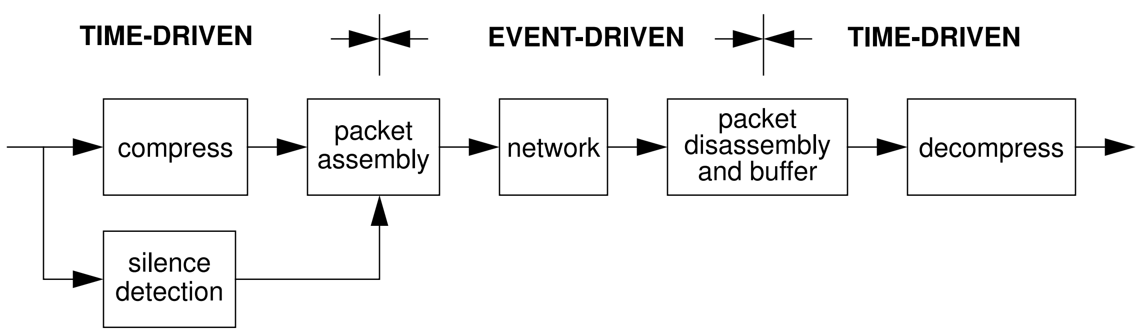


Figure 4.1 A packet speech system simulation. The signal processing portions of the algorithm (compression, silence detection) suit a time-driven model, while packet assembly, disassembly and transport are best modelled using a discrete-event simulation model. Figure from [Ha92].

two regions of space, or even two distinct "universes". In some speculative cosmological models, such as the original inflation model of Guth, distinct laws of physics may operate in the two connected "universes" (corresponding to symmetry breaking in different "directions"), and these separate regions are called domains.[1] Accordingly, we adopted these terms for use in describing related terms in Ptolemy (Ptolemy is named after a famous astronomer because of the use of astronomical terms in the system). Briefly, a wormhole is an object that appears to be a star belonging to one domain from the outside, but on the inside, contains a galaxy, scheduler, and target object appropriate for a different domain.

4.1.2 The Organization of Ptolemy

In Ptolemy, every functional block is derived from the basic class **`Block`**. A block may contain one or more inputs and outputs known as *portholes*, which are objects derived from class **`PortHole`**. Portholes permit blocks to connect to other blocks and permit messages to be transmitted between them; these messages are objects derived from class **`Particle`**. The basic atomic actor, **`Star`**, and the basic composite actor, **`Galaxy`**, are both derived from class **`Block`**.

The link between two connected portholes is implemented by the class **`Geodesic`**. The class **`Plasma`** implements a pre-allocated pool of **`Particle`** objects to avoid expensive particle allocation and de-allocation at run time. The connection between Blocks in a typical simulation model is shown in figure 4.2.

For each domain, there is a corresponding star class and porthole class; for example, for the hypothetical domain **`XX`** we would have **`XXStar`** and **`XXPortHole`**. Thus actors belonging to the SDF domain are derived from class **`SDFStar`** and actors belonging to the DE domain are derived from class **`DEStar`**. Each of these classes is in turn derived from class **`Star`**. We do not require a different derived type of galaxy for each domain; the domain of a galaxy is determined by the objects it contains and for most purpose the **`Gal-`**

1. Interested readers are referred to Hawking ([Haw88]) for a description of these theories that is accessible to the non-physicist. It should be noted that Ptolemy does not attempt to be "astrophysically correct" in the use of these terms; they are only suggestive.

axy class merely serves as a means for introducing hierarchy.

Wormhole objects are implemented using multiple inheritance, meaning that there is more than one base class for the object and that it implements the interface required for each of the base classes. For example, an object of class **SDFWormhole** is multiply inherited from class **SDFStar** and class **Wormhole**; and in general **XXWormhole** is derived from class **XXStar** and the **Wormhole** class. The Wormhole class cannot be used alone; it has a scheduler, a target object, and a galaxy, all of which correspond to the "inside" of the wormhole. A portion of the class inheritance hierarchy is shown in figure 4.3.

The class of the wormhole object (e.g. **SDFWormhole**) corresponds to the outer

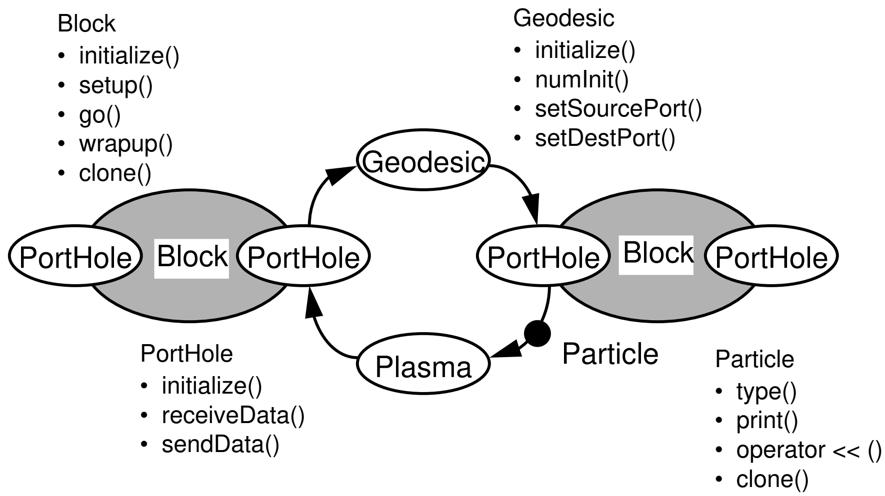


Figure 4.2 Block objects in Ptolemy send and receive data encapsulated in Particles to the outside world through Portholes. Buffering is handled by the Geodesic; recovery of used Particles is handled by the Plasma (from [Buc93b])

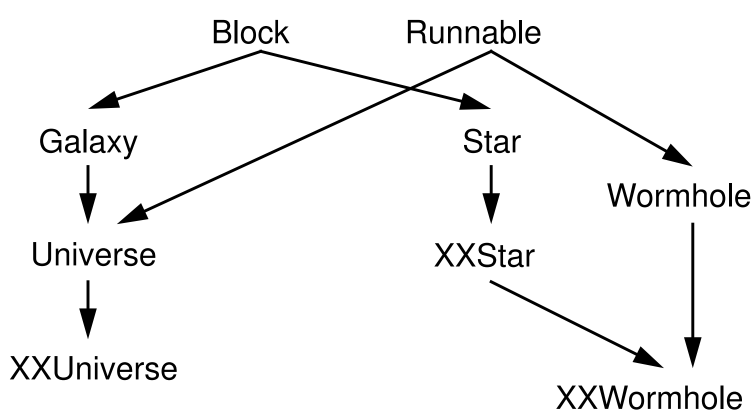


Figure 4.3 A portion of the inheritance hierarchy for blocks and wormholes in the hypothetical domain XX.

domain of the wormhole object since the class of an object determines its external interface; the inner domain of the wormhole is determined by the types of objects it contains. The principle that a wormhole looks like a star from the outside must be remembered by the wormhole designer and is the key to permitting differing domains to interface cleanly. From the graphical interface, the user creates wormholes by causing an instance of a galaxy belonging to one domain to appear as a member of a galaxy or universe of another domain; this means that to the casual user, domain boundaries look much like galaxy boundaries.

The portholes of a wormhole object are special because they perform a conversion function on messages traversing the domain boundary. Because of this difference, the boundary between the input and output of a wormhole is implemented by a special object known as an *event horizon*.[1] The conversions required at the event horizon are domain-specific. One possible approach to interfacing different domains would be to provide a separate type of event horizon for each pair of domains interfaced. Unfortunately, as new domains are added the expense of this technique would grow as the square of the number of domains. Instead, our approach is to convert particles crossing domain boundaries to a "universal" representation, and thereby implement objects that convert signals from each domain to and from this representation. This requires $2N$ conversion methods instead of N^2 methods, one to convert from each domain to the universal representation, and one to convert from the universal representation to each domain-specific representation. We cannot guarantee that this approach will be successful for all possible domains, so it may still be considered experimental.

Event horizon objects are implemented using multiple inheritance in much the same way as wormholes are. For the domain **`XX`**, we have the classes **`XXtoUniversal`** and **`XXfromUniversal`**. The former is derived from the classes **`XXPortHole`** and **`ToEv-`**

1. For a black hole, an event horizon is the boundary of the region from which nothing, even light, can escape. It is arguable that the terms "event horizon" and "wormhole" in Ptolemy should be reversed.

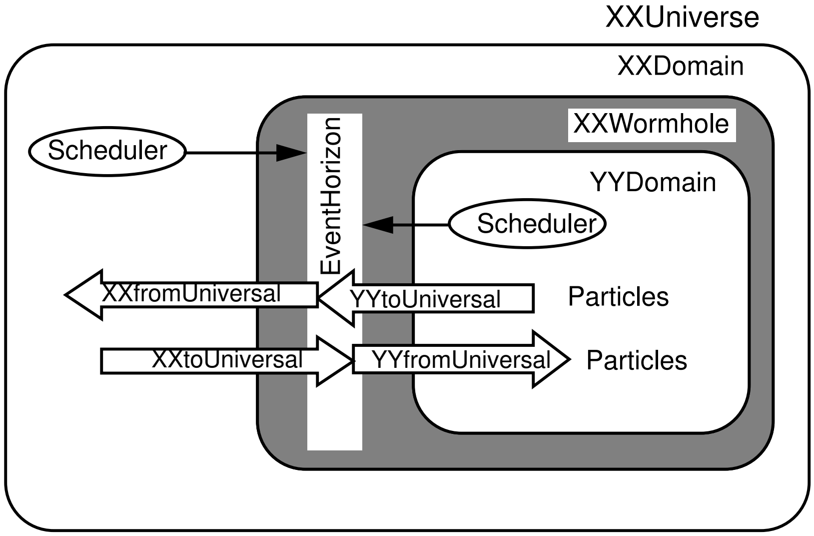


Figure 4.4 The event horizon interfaces two domains and converts particles from the representation required in one domain to that used in another.

`entHorizon`; the latter is derived from **`XXPortHole`** and **`FromEventHorizon`**. **`ToEventHorizon`** and **`FromEventHorizon`**, in turn, are derived from **`EventHorizon`**. The wormhole object contains a pair of event horizon objects for each connection that traverses the wormhole boundary, one to convert from the inner domain to the universal representation and one to convert from this representation to that of the outer domain (for connections travelling in the opposite direction, the event horizon objects are arranged to perform the reverse conversion).

Conversions commonly needed at wormhole boundaries include the generation, removal, or conversion of time stamps associated with **`Particle`** objects. It may also be necessary to transmit data to a different process or processor; in this case, methods of the **`Target`** object associated with the wormhole are used to perform the inter-process or inter-processor communication. A detailed discussion of the wormhole/event horizon interface may be found in [Buc93b].

Ptolemy simulations execute under the control of *schedulers*. The top-level universe object contains a scheduler; so do any wormholes the simulation contains. In some domains (such as SDF), the entire order in which blocks are to be executed may be determined by the scheduler's setup method; in other cases (such as DE), the scheduler's operation is highly dynamic, and the order of execution is determined at run time. For code generation models, an object derived from the class **`Target`** represents the target of code generation. The scheduler acts as the slave of the target object and is used to determine the order of actor executions, while the target controls such operations as downloading code, and required inter-process or inter-processor communication, and so forth. In simulation domains, the target is still present but mostly passes commands on to the scheduler (it may, however, be used to select among several schedulers or to pass parameters that control the scheduler's operation).

4.1.3 Code Generation in Ptolemy: Motivation

Practical signal processing systems today are rarely implemented without some form of software or firmware, even at the ASIC (application-specific integrated circuit) level. Programmable digital signal processing chips (PDSPs) form the heart of many implementations. For tasks that are computationally demanding, even the fastest PDSPs are not sufficiently powerful, so some custom circuitry is often required. A new implementation technology that is now available from several major manufacturers of PDSPs is *DSP cores*. A DSP core is a programmable architecture that forms only a portion of a single integrated circuit, unlike standard PDSP chips that are separate components. Thus a designer can produce an ASIC that is equivalent in function to a circuit board containing a standard PDSP chip and custom circuitry. Such devices are already being used extensively for digital cellular telephony [Pin93].

The task of designing an ASIC that uses a DSP core resembles the problem of designing a circuit board; it requires a mixed hardware and software design. Thus any complete system design methodology for DSP applications must include software synthesis; and accordingly commercial manufactures of DSP development environments, such as Comdisco Systems, Mentor Graphics, and CADIS, have recently added such capabilities [Pow92][Rab91][Des93].

It is desirable to be able to simulate the software and the hardware portions of the system together, and to cleanly support heterogeneity since the design styles and modelling for the different portions of the system can be expected to be very different. Ptolemy was designed from the beginning to support this kind of heterogeneity.

It is, of course, possible to program PDSPs in a high level language such as C, however, features of most PDSPs are not well-modeled by C or other conventional high level languages, so that code produced by most C compilers has not been satisfactory to many designers. More specialized DSP languages such as Silage, an applicative language with fairly simple semantics, have also been used, for example in the DSPStation application from Mentor Graphics [Gen90]. In Ptolemy we use a third alternative, one adopted from the Gabriel system. In this model, actors generate small pieces of hand-written assembly language corresponding to functional operators. Actors may be fine-grain or coarse-grain, and may possess state (actors with state place extra constraints on the scheduler but are otherwise cleanly handled). There are two phases to code generation under Ptolemy (or Gabriel): scheduling and synthesis. The scheduler possibly partitions the actors for parallel execution and determines their order. The synthesis phase stitches the hand-written code segments (which may be assembly language, a higher level language, or a mixture) together. This technique has been commercialized by Comdisco (see [Pow82]), CADIS, and others, although there are important differences between their techniques and ours; for a detailed discussion, see [Pin93].

4.1.4 Targets and Code Generation

The **`Target`** class was first conceived of to model the environment for which code is to be generated by Ptolemy, and it is the **`Target`** that ultimately controls the execution of any Ptolemy process, whether it involves simulation, code generation, or a combination of both.

For code generation applications, the **`Target`** defines how the generated code will be collected, specifies and allocates resources such as memory, and defines code necessary for proper initialization of the platform. The **`Target`** will also specify how to compile and run the generated code. Optionally, it may also define wormholes. A **`Target`** may represent a single processor or multiple processors; in the latter case, the interconnection network is also specified.

All code generation targets are derived from the base class **`CGTarget`**, which defines methods for generating, displaying, compiling, and executing code (as is standard in object-oriented design, derived classes may accept these default methods or replace them with domain-specific methods, as appropriate). There are derived classes **`AsmTarget`** for assembly language code generation (which adds methods for the allocation of physical memory) and **`HLLTarget`**, the base class for synthesis of high-level-language code (such as C). Targets for the generation of a specific kind of assembly language would be derived from **`AsmTarget`**, (e.g. **`CG56Target`** permits the generation of assembly language code for the Motorola 56000), and targets for the generation of a specific high-level language would be derived from **`HLLTarget`** (e.g. **`CGCTarget`** for C code).

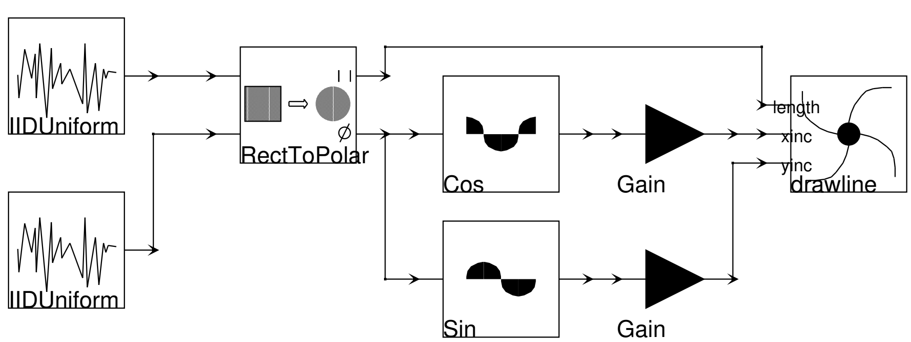
In code generation applications, rather than computing particular results, stars are designed instead to produce code that computes these results. Schedulers are responsible for determining the order in which these actors will be executed, and Targets collect, download, and execute the resulting code. In the current implementation, stars always communicate through memory, and memory buffers are allocated for each arc. Future implementations will permit assembly language stars to communicate through registers instead. To reduce the number of copy operations, Ptolemy supports a “fork buffer” mechanism that permits the input and all the outputs of a FORK actor to share the same buffer, and an “embedded buffer” mechanism that, in some cases, permits actors such as DOWNSAMPLE to be implemented without any code (the output arc of the actor corresponds to one memory location inside the buffer for the input arc).

Code generation from regular dataflow graphs in Ptolemy is described in detail in [Pin93].

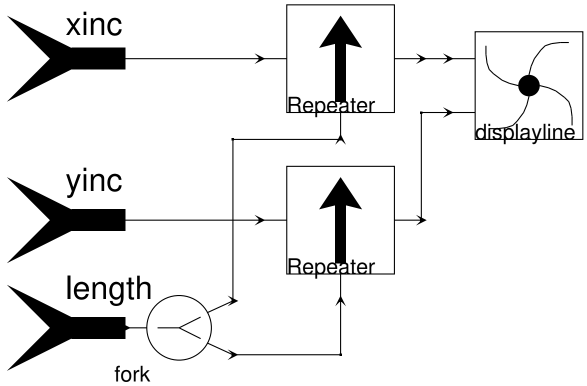
4.1.5 Dynamic Dataflow In Ptolemy: Existing Implementation

Ptolemy contains both an SDF domain, which is restricted to regular dataflow, and a DDF domain, which permits any type of dynamic dataflow actor. Since an SDF actor is a special case of a DDF actor, the implementation uses a common base class, **`DataFlowStar`**, and the DDF scheduler is able to execute any actor that is a member of this class.

Given a dataflow application that contains some data-dependent decision-making, and given the greater efficiency that can be achieved with SDF, one approach that naturally suggests itself is to group, as much as possible, those portions of the dataflow graph that are regular into separate wormholes, so that large portions of the graph can be scheduled statically. One way to do this is to ask the user to do it manually, by grouping subsystems together in galaxies and marking all the galaxies that contain only regular actors as SDF. By means of nesting, in which DDF and SDF domains are alternated, the amount of run-time scheduling required can be reduced considerably. This was the first approach taken in Ptolemy's development. As a simple example of this approach, consider the following Ptolemy program, intended to suggest the path taken by a moth. At the top level, we generate a random sequence of direction vectors, with the following program, which is a regular (actually homogeneous) dataflow graph:

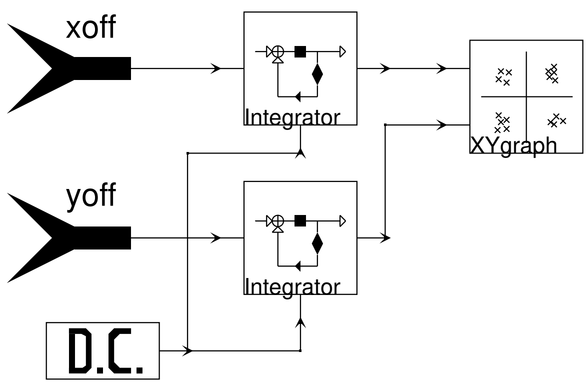


All of the actors except for "drawline" are primitive actors. The "drawline" actor accepts a length value, which is converted to an integer, and a "unit vector", supplied by the inputs "xinc" and "yinc". This actor, when executed, will add "length" points to the graph, using the vector (xinc,yinc) as the offset between points. If we expand the "drawline” actor we see the following:



Here the “Repeater” actors are not regular dataflow actors. The bottom input, an integer, specifies the number of output tokens produced; each output token is a copy of the input token[1]. The number of tokens to be produced on the output arc of this actor is not known until the graph is executed. The “displayline” actor adds a single line to the graph, given the X and Y coordinates of the relative motion from the input vector.

Finally, the interior of the “displayline” actor appears as follows:



This graph is, once again, a regular (homogeneous) dataflow graph. The “Integrator” actors form a running sum of their inputs, and the “XYgraph” actor adds each input pair to the graph. By setting the domain of the innermost and outermost levels to “SDF” and the domain of the “drawline” galaxy to “DDF”, Ptolemy constructs wormholes in such a way that dynamic scheduling is only required to run “drawline”; otherwise, static

1. The Repeater actor is not a BDF actor as the number of tokens produced depends on an integer-valued control token; extension of the BDF model to support such actors directly is discussed in Chapter 5. It is possible, however, to represent a Repeater actor using a do-while loop of BDF actors.

scheduling is used.

It is also possible to apply clustering methods to group adjacent regular actors together, effectively creating the same type of partitioning we might otherwise require the user to perform. We then have several choices about what to do with the clusters and dynamic actors that remain. One possibility is to simply execute them completely dynamically using a general dynamic dataflow scheduler, while using a schedule generated at compile time for each cluster. This is reminiscent of the hybrid dataflow techniques discussed in section 2.3. Another is to attempt to recognize certain standard "dynamic constructs" such as if-then-else (or the more general case statement) and do-while and treat them specially. If the entire graph can be so classified, it is then possible to generate code using Lee's quasi-static scheduling idea [Lee88a]. This approach is explored in detail in [Ha92]. In Ha's work, rather than finding constructs by means of the token-flow analysis and clustering techniques of chapter 3, a more limited pattern-recognition approach was used. This approach is sufficient in many cases to recognize the constructs, especially in graphs where there is very little use of dynamic actors.

4.2. SUPPORTING BDF IN PTOLEMY

We now describe the implementation of Boolean-controlled dataflow (BDF) under Ptolemy. Adding the new domain required some re-thinking of the Ptolemy class hierarchy to permit better sharing of code between SDF, BDF, and DDF domains and to simplify use of the BDF model for code generation.

The design goals for the project were as follows: we wanted to support BDF models of execution both for simulation and for code generation. The simulation model should be able to generate clusters of actors that are scheduled statically, and, if clustering cannot completely succeed, execute the resulting clusters dynamically, as described in section 3.4.3. It should be possible to use BDF simulation actors under the existing DDF scheduler, as well as existing SDF simulation actors under the BDF scheduler. All single-processor code generation targets should be able to support BDF code generation actors and constructs. We did not address parallel scheduling of BDF actors in this project; that is an area for future research.

So that the new actors fit conveniently into the existing design, we clearly wish for **BDFStar** to be derived from **DataFlowStar**; this means that the DDF scheduler can execute BDF stars as well (as it should, since BDF actors form a subset of dynamic dataflow actors). However, we wish to have a BDF scheduler successfully execute objects of class **BDFStar** as well as **SDFStar**, but not **DDFStar**. Given this consideration, one possibility would be to introduce a common base class for **BDFStar** and **SDFStar**. However, the situation becomes more complex when code generation stars are also considered. Under the initial Ptolemy implementation, stars that generate assembly language code for the Motorola 56000 DSP chip using regular dataflow semantics form the CG56 domain and are derived from **CG56Star**, and **CG56Star** was derived from **SDFStar** (indirectly). How should dynamic actors for the 56000 be implemented? Should they form a separate domain and be derived from **BDFStar**? This would require large amounts of code duplication and other difficulties in implementing the portions of the code and behavior common to all stars that generate the same language, although it might be possible to resolve with multiple inheritance. Unfortunately, because of some inconvenient features of the C++ language, this type of solution was considered too expensive and complex, and was rejected.[1]

We therefore changed the class hierarchy so that all code generation stars are derived from **DataFlowStar** but not from **SDFStar**. The class **DataFlowStar** has a virtual function **isSDF()** which returns TRUE if the object on which it is called obeys SDF (regular dataflow) semantics and FALSE if it does not. The default implementation of methods in **DataFlowStar** correspond to those for a regular (SDF) dataflow actor. Schedulers that require regular dataflow semantics on their actors must now call **isSDF** to

1. For a detailed discussion of the impact of the features of the C++ language on Ptolemy's design, see [Buc91c].

test that the actors obey the required semantics.

Just as we have a common base class for the stars, we also have a common base class for the portholes. Classes **`SDFPortHole`**, **`BDFPortHole`**, and **`DDFPortHole`** all have a common base class, called **`DFPortHole`**. The base class has virtual functions that specify whether the number of tokens transferred per execution is fixed or varying. There is also a method that returns the number of tokens transferred on each execution by default; for non-varying portholes, this is the number that is always transferred. In addition, virtual functions are provided that permit the porthole to indicate that another **`DFPortHole`** is "related" to the given **`DFPortHole`** (the associated port), as well as to return a code indicating the nature of the relationship. This feature is used by BDF portholes to indicate, for example, that another port is the control port for this port. There are currently five possible relations, with provisions for extension:

- DF_TRUE: the port transfers data only when the token from the associated port (the control port) is TRUE;
- DF_FALSE: the port transfers data only when the token from the associated port is FALSE;
- DF_SAME: the stream produced on this port is logically the same as the stream on the associated port (this relation is used for fork actors, for example);
- DF_COMPLEMENT: the stream produced on this port is the logical complement of the stream on the associated port (this relation could be used by a logical NOT actor).
- DF_NONE: there is no specified porthole relationship at all.

Because of the structure of these relations, we impose some restrictions on the actors we can represent. In section 3.1, we required only that the number of token transferred by a conditional port be a two-valued function of a control Boolean. We now require that one of the two values be zero. Furthermore, we currently do not provide a way to model certain relationships; for example, we do not represent information sufficient to reason about cases where Boolean streams are subsampled by SWITCH actors in such a way that two subsampled streams are equivalent (as is discussed in the section “Mutually Dependent Booleans” of [Lee91b]). It did not appear that there was sufficient payoff from the added complexity, although as a result, some unusual graphs that are in fact strongly consistent may be reported as weakly consistent, and some graphs with a bounded state space may appear to be unbounded. In practice, these restrictions have not proved to be a problem, although our experience is still limited.

The class hierarchies for dataflow stars and portholes resulted in two isomorphic trees. All star classes, as stated, are derived from **`DataFlowStar`**, and all porthole classes are derived from **`DFPortHole`**. From these are derived the classes **`SDFStar`** and **`SDFPortHole`**, respectively, representing simulation objects obeying regular dataflow semantics. The classes **`DynDFStar`** and **`DynDFPortHole`** are the base classes for all other stars and ports, respectively, and contain some support for execution under dynamic schedulers. **`BDFStar`**, representing BDF simulation actors, is derived from **`DynDFStar`**, as is **`CGStar`**, representing all code generation stars. The latter derivation provides support for BDF semantics in all code generation domains. The derivation tree for portholes is analogous (see figure 4.5).

We permit clusters of BDF simulation actors to be executed by a dynamic scheduler, but we do not support dynamic scheduling of code generation stars (other than in the sense that generated if-then-else or do-while constructs constitute dynamic scheduling). Accordingly, schedulers are designed to “inform” stars (by calling the **`setDynamicExecution`** method of **`DataFlowStar`**) whether they will be executed by a dynamic scheduler; class **`CGStar`** will report an error in such cases indicating that the operation is not supported.

4.3. STRUCTURE OF THE BDF SCHEDULER

The BDF scheduler performs a limited version of the “strongly consistent” check

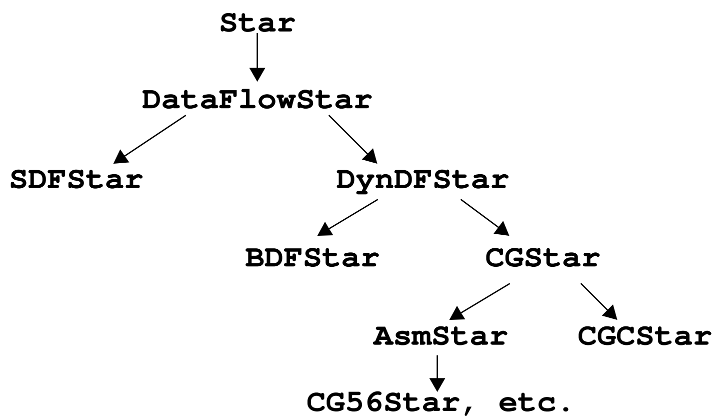


Figure 4.5 Inheritance hierarchy for dataflow and code generation stars. The hierarchy for portholes has the same form, with class names obtained by substituting PortHole for Star (except DataFlowStar -> DFPortHole).

on graphs presented to it, followed by loop scheduling and, if necessary, state space traversal. At this point, state space traversal has not yet been implemented, other than the simple form necessary to recognize certain types of do-while loops; we will discuss the planned implementation strategy, however.

4.3.1 Checking For Strong Consistency

The check for strong consistency proceeds by associating an object called a **BoolFraction** with each actor and computing its value, in much the same way as a regular dataflow scheduler computes the repetition value of each actor. A **BoolFraction** has a numerator and denominator, each of which is a **BoolTerm**; a **BoolTerm** has a constant term plus a list (possibly zero length) of **BoolSignal** objects. A **BoolSignal** object contains a reference to a control signal and a desired value, which is either TRUE or FALSE. A **BoolTerm** can be considered to be a product of its constituent constant term and **BoolSignal** terms. Given this representation, we can now compute the repetition vector for the system. At this stage we consider only equality of long-term rates, so a control signal is considered equivalent to a delayed version of itself. We begin by picking an actor and assigning it a repetition rate of one (represented by a **BoolFraction** with numerator and denominator both one). Each adjacent actor that has not had its repetition

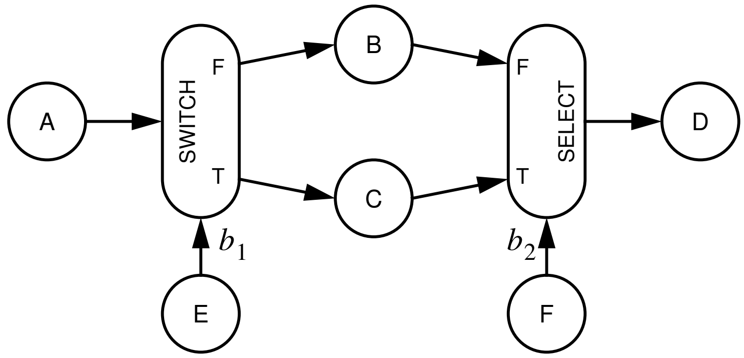


Figure 4.6 A weakly consistent graph, used as an illustration of consistency failure detection. All actors other than the SWITCH and SELECT are homogeneous.

rate set yet is assigned an appropriate value to solve the balance equations; if there are cycles in the graph when considered as a nondirected graph, a given actor will be reached more than once, at which point a consistency check is performed. If two different paths to an actor determine two different repetition rates, an error results. This algorithm is exactly the same as the one described in section 2.2.1 for regular dataflow graphs.

To report a useful error to the user, any common factors in the two BoolFractions are eliminated and what remains is reported as an error. We then obtain a diagnostic message like

```
Consistency failure detected at Select1:
Select1.control != Switch1.control
```

for the graph in figure 4.6.

4.3.2 Clustering BDF Graphs: Overview

The BDF loop scheduler is responsible for implementing the clustering algorithm described in section 3.3.3. It does so by constructing a parallel hierarchy of clusters corresponding to the dataflow graph it is presented with, and by successively transforming this group of clusters by applying merge operations, which merge actors with the same repetition rate into single clusters where possible, and “loop” operations, which introduce conditionals and loops into the graph. The most complex part of the implementation has to do with constructing the relationships between the ports of the cluster actors (e.g. DF_TRUE and DF_FALSE to indicate conditional ports, and DF_SAME to indicate ports with the same value) and keeping them consistent. This is simply a matter of careful bookkeeping, however.

The abstract class **BDFCluster** represents a cluster. There are three kinds of cluster, each derived from **BDFCluster**: **BDFAtomCluster**, which corresponds to a single actor in the original graph, **BDFClusterBag**, a composite cluster with an internal schedule, and **BDFWhileLoop**, a special type of composite cluster that represents a do-while loop. A cluster has a set of input and output arcs (class **BDFClustPort**), a loop count (which may indicate that the contents are to be executed n times, for some n), and an optional condition (which indicates that the cluster is only to be executed if some control token has either a TRUE or a FALSE value).

The top level of the clustering algorithm is simple to describe: first a “cluster galaxy” consisting of one **BDFAtomCluster** for each actor from the original universe is built. We then alternate two passes, called the merge pass and the loop pass, until no further transformations can be made. An internal schedule is computed, using regular dataflow methods, for each composite cluster. Because each cluster consists only of actors with the same repetition rate, these schedules have a very simple structure: they are data-independent, and each subcluster will be executed exactly once. All data dependencies are represented either by the inserted “if-then-else” or “do-while” constructs, or remain visible at the top level.

“Looped” clusters will have if-then-else or do-while constructs around them, or else will have a constant repetition factor. At this stage, some clusters may have multiple subclusters that are conditionally executed based on the same condition, or on opposite values of the same condition; a merge pass is run at this point to combine them into larger loops and into if-then-else statements.

If the top level is reduced to a single cluster or is a regular dataflow graph, we compute a schedule for the top level and we are done. If not, and this is a simulation run rather than a code generation run, we can execute the top-level clusters with a dynamic dataflow scheduler, treating each cluster as a single actor.

4.3.3 The Merge Pass

The goal of the merge pass of the BDF scheduler is to transform the input BDF graph into a new BDF graph by combining adjacent actors into a single cluster, in such a way that each cluster will have a static, data-independent internal schedule. In order to merge two adjacent actors, several conditions must be met. It should be noted that these conditions are sufficient but not necessary.

First, we retain the conditions that pertain to cluster merging in regular dataflow graphs; these are described in detail in section 3.3.2. Consider a pair of adjacent actors we wish to merge, consisting of a source actor S that produces tokens on an arc and a destination actor D that consumes tokens from the same arc. Briefly, the merged actors must have the same repetition rate and merging them must not cause deadlock, which may occur if there is a path from the source actor to the destination actor that passes through a third actor.

In addition, we obtain more conditions, imposed by the requirement that the new graph we obtain by the merge operation must also be a BDF graph and that the internal schedule be data-independent. We must avoid "burying" control arcs: if any of the arcs that connect S and D have control ports for conditional ports of either S or D that will remain external ports after S and D are merged, we may not perform the merge unless the control ports can be "remapped", or if the merged cluster can be turned into a do-while loop with the correct semantics. Remapping of control arcs and the creation of do-while loops is described later.

Normally, all arcs that connect the actors that are merged become internal arcs, not visible from the exterior of the cluster. There are two exceptions: first, if the control arc that would be buried contains initial tokens, we permit the merge and transform the control arc into a self-loop of the merged cluster (the merge is permissible in this case because the control arc remains visible). Second, to assure the data-independence of the internal schedule, arcs with mismatched control conditions at either end will be also be transformed into self-loops. An example in which both of these types of self-loops are created appears in section 3.3.3 on page 90.

This is a complex set of conditions that may require repeated searching of the entire graph for paths. Fortunately, in most cases it can quickly be determined whether two actors can be merged based only on local information. If all outputs of S connect directly to D, or if all inputs to D connect directly to S, and there are no initial tokens on at least one arc, then merging cannot possibly create deadlock. Since most dataflow actors have only one output, only one input, or both, this is a common case. Furthermore, most arcs are not control arcs. Therefore the merge pass consists of a “fast part” that merges as many pairs of adjacent actors as possible without performing any path searches or control arc remapping, followed by a “slow part” that searches for indirect paths and remaps control arcs where possible and necessary (after the size of the graph has already been reduced by the application of the fast part).

Remapping of control arcs is accomplished by exploiting DF_SAME and DF_COMPLEMENT relations on arcs. FORK actors have an indication that all arcs provide the same signal, and other actors may be designed to provide this indication as well. For the NOT actor, the output arc is marked as being the complement of the input. If an important control signal would be buried by merging two actors, but the same signal is available via a DF_SAME relation on an arc that will remain external, the merge operation may proceed anyway and the porthole relations in the new cluster are remapped to use the signal that remains external. To ease the operation of remapping control arcs, the class **`BDFClustPort`** possesses an iterator mechanism that sequentially steps through every arc that can be considered the same as, or the complement of, a given arc, so that this complex operation need be implemented only once.

4.3.4 The Loop Pass: Adding Repetition

It is the task of the loop pass to transform clusters of the dataflow graph to enable subsequent merge passes to combine more clusters. To do this it must alter the clusters in such a way that their repetition rates will match those of their neighbors. Three transformations of a cluster are possible: a cluster may be repeated for a fixed number of times, a cluster's execution may be made conditional on some control token, or a do-while loop may be added around a cluster (so that the cluster is executed repeatedly until a desired value appears on some control arc). Two of these three transformations cause control loops to be added to the execution of the graph, hence the name "loop pass."

The first transformation, corresponding to iteration of a cluster a fixed number of times, is easiest to describe. There are two cases: *integral rate conversions*, in which the number of tokens transferred at one end of an arc evenly divides the number of tokens transferred at the other end, and *nonintegral rate conversions*, in which this condition does not hold. These cases are handled exactly the same way as they are for regular dataflow graphs in the algorithm described in section 3.3.2. The only additional considerations are these: we do not loop a cluster to match the rate of its neighbor by inserting a constant loop factor if there is also a difference in control conditions (one end of the arc is conditional but the other is not, or the two ends are controlled by different conditions). Only "if" conditions and "do-while" loops may be inserted in such cases. Second, in regular dataflow graphs certain graphs with feedback loops containing delays can be looped given knowledge of the repetition count of each actor (arcs with "enough delay" can be completely ignored, as discussed in [Bha93b]); these techniques are not applicable for BDF graphs so clustering must sometimes be more conservative.

4.3.5 The Loop Pass: Adding Conditionals

In addition to adding repetition to cause a cluster to match the rate of its neighbors, the loop pass may also add conditionals. Given an arc where one end transfers tokens conditionally and the other end transfers tokens unconditionally, and the constant term is the same (e.g. actor A always produces 2 tokens, actor B consumes 2 tokens if its control port provides a TRUE token) we have a possible candidate for making a cluster conditional. In many cases, if a cluster is made conditional we must add an extra arc that serves to pass the conditional token from its source to the cluster that requires it. An example of this appears in section 3.3.3. To accommodate this, the implementation provides a mechanism for creating duplicate arcs to pass conditions from one cluster to another.

If the control arc is on a self-loop, we may wish to avoid creating a conditional construct so that a do-while may be created instead. Consider the example in figure 4.7, which might arise in the process of clustering a system with data-dependent iteration. It would be possible to add an "if" around clusters 1 and 5, and then merge them into the main cluster. We would then add a "while" around the whole system. But then actors 1 and 5 would both appear inside both an "if" statement and a "while" statement, even though they are each executed exactly once. For now, we avoid creating a conditional construct if the if-condition matches the state of the initial token on the feedback arc, since this means that a "do-while" form of clustering is likely to succeed.

4.3.6 Loop Pass: Creation of Do-While Constructs

Do-while constructs, in which a cluster is repeatedly executed until a control token with a particular value is produced on some control arc, may be created in either of two ways. The first possibility is that an actor that contains a control signal on a self-loop may, if conditions are right, be transformed by adding a do-while condition around it. The second possibility is that a pair of adjacent actors, in which one produces a control signal and

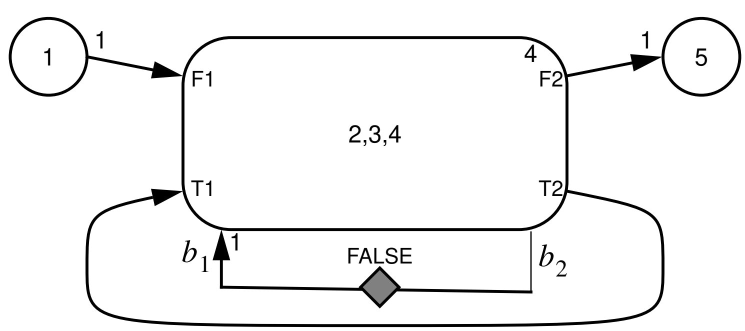
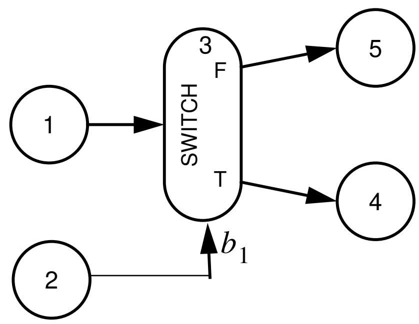


Figure 4.7 A partially clustered do-while system. At this point, it would be possible to make either actor 1 or actor 5 conditional so that a subsequent merge pass can combine them with the main cluster. We prefer to put only the main cluster inside the while loop to more accurately reflect the control structure.

one consumes it, may be simultaneously “merged” and “looped” to produce a do-while loop.

There is a natural tension between the creation of an “if” construct and the creation of a “while” construct. In many dataflow graphs, it is possible to create either type of construct in the process of obtaining a complete clustering. Consider the following graph:



It is identical to figure 3.7 except that actor 5 is now homogeneous. Clearly we could cluster this graph by the introduction of conditionals, obtaining a schedule like

```
(1,2,3,if(cond) then 4 else 5)
```

But the clustering we obtained for figure 3.7 in section 3.3.3 would work as well; in this case, we would obtain

```
do { 1,2,3, if(cond)4} while cond; 5
```

We could also obtain the alternative clustering

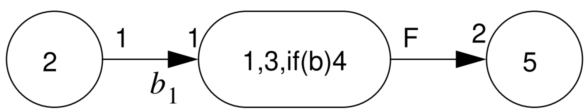
```
do { 1,2,3, if(!cond)5 } while !cond; 4
```

Our implementation favors the creation of “if” over “while” where possible as it leads to bounded-length schedules where they exist. It is possible that one of the latter schedules may be preferable in some circumstances. The third schedule would be preferable, for example, if the task is to repeatedly execute the graph until actor 4 has been executed some number of times.

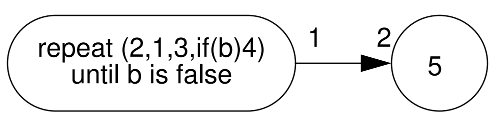
If a pair of actors meet all the conditions for merging other than that the merge operation would bury a control arc, it is possible that the pair may still be merged by the creation of a while loop. The while loop will have the form

```
do {S; token = S.control; D} while(token == value)
```

The code for insertion of do-while loops determines whether the insertion of this type of loop is legal. For it to be permissible, all arcs of the source and destination actors that remain external after the merge must be conditional on the control signal, and conditional in the same way; they will become unconditional after the addition of the while loop and the direction of their conditionality will determine the termination condition. For example, in the process of clustering the example in figure 3.7 we obtained the following intermediate clustering:



Here we are considering merging actor 2 with the cluster at the center. There will then be one external port, and it is indeed conditional on the control signal that connects the actors to be merged. The fact that it produces output when the signal is FALSE determines the sign of the loop termination condition: the loop executes until a FALSE token is produced. Since there will be exactly one FALSE token, the conditional goes away, and we obtain the clustering given below:



The second type of do-while loop is created from a single actor or cluster. This single actor always has one or more Boolean control signals in the form of self loops, so that the same actor both produces and consumes the control signal. Such actors may possess other self loops as well. In order to create a do-while in this circumstance, all external ports must be conditional with the same sign (TRUE or FALSE) and depend either on the same signal or on delays of the same signal. Dependence on delays is permissible because the loop will generate one cycle of both the signal and all of its delays. After the while loop is added, the new cluster will be unconditional.

If the actor we started with contains self-loops, or the pair of actors we started with has additional conditional arcs that pass between the actors to be merged, we must also check that the loop created will be "proper", in the sense that it is bounded. Doing so forms a simplified case of the general state traversal problem. We currently handle only the cases in which there are exactly two states, one corresponding to the production of a TRUE token on the control arc and one corresponding to the production of a FALSE token. Other cases are objected to by the current implementation.

4.4. GRAPHS LACKING SINGLE APPEARANCE SCHEDULES

Some BDF graphs cannot be completely clustered, so that after the clustering algorithm has completed its work, multiple clusters are left at the top level and the top-level graph is not a regular dataflow graph. In some cases, even though the graph cannot be reduced to a regular form, a static schedule (one consisting only of sequential execution, fixed-repetition loops, if-then-else constructs, and do-while constructs) may sometimes still exist. This happens for BDF graphs that, in the terminology of [Bha93b], lack single appearance schedules. One such example appears in figure 4.8. To avoid having a complicated maze of wires, the graph has been simplified; actor E's output is connected to a six-way FORK actor that passes identical control streams to each of the six dynamic actors (three SWITCH and three SELECT actors). This graph has an interesting property: based on local information, each connection appears to be "logically homogeneous" in that the source star produces exactly what the destination star consumes. Despite this, the order of execution must depend on the particular Boolean values generated.

The graph can be partially clustered; actors B and C may be combined with the SWITCH and SELECT actors that are adjacent to them, as can actors A and D. Furthermore, the actor E and the FORK can be combined with the cluster containing actor A. The resulting graph cannot be clustered further, but nevertheless the graph can be shown to have a bounded-length schedule. In fact, the following schedule executes the graph cor-

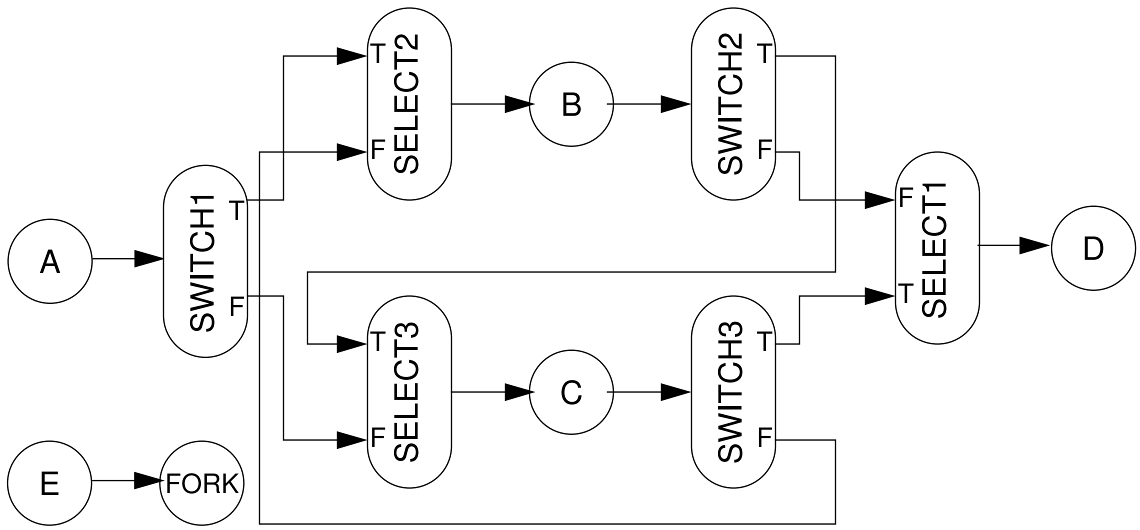


Figure 4.8 A BDF graph that lacks a single appearance schedule. Certain arcs have been omitted from the graph to make it easier to understand: the FORK actor connected to E, whose outputs are not shown, passes the stream of tokens from actor E to the control input of each of the SWITCH and SELECT actors. Crossing lines do not imply a connection.

rectly:

```
A,E,FORK, SW1;
if(E.output) {SEL2,B,SW2,SEL3,C,SW3}
else {SEL3,C,SW3,SEL2,B,SW2}
D
```

Note that six actors appear in the schedule twice. The total number of times each actor is executed in the schedule is exactly one, but since the order of execution and data dependencies depend on the value of the Boolean token, we do not have a data-independent schedule. Our current implementation does not generate such schedules, though they can be generated by a process we call "node splitting" that has been designed but not yet implemented.[1] It is applicable for graphs with bounded-length schedules that cannot be completely clustered. We compute the repetition vector, considering it as a list of tasks to be performed, and attempt a topological sort. In the above example, each actor is to be executed once. We succeed in scheduling A, E, FORK, and SWITCH1, but find that the

1. In the literature of optimizing compilers for procedural languages, "node splitting" refers to a process of code duplication that converts unstructured code with many "gotos" to a structured form. This procedure was first described in [All72].

SELECT2 actor can only be executed conditionally. We therefore split it into two separate tasks to be performed, corresponding to **`if(E.output)SEL2`** and **`if(!E.output)-SEL2`**. We then find that we can schedule the former task. We proceed in this manner, splitting nodes into two tasks only when otherwise no actors can be executed, until all actors have been scheduled the number of times required. This operation succeeds in scheduling any graph that has a valid bounded length schedule, but in code generation applications, code size may increase considerably unless subroutine calls are used to avoid duplication of code.

4.5. MIXING STATIC AND DYNAMIC SCHEDULING

If, after clustering, the resulting BDF graph does not have a bounded length schedule and we must execute the graph anyway, dynamic scheduling is required, together with dynamic memory allocation on certain arcs. When executing dataflow graphs in the simulation environment, this requirement is not a problem; it is already supported for general dynamic dataflow actors. To be as efficient as possible, we wish for the clusters to be considered atomic actors from the point of view of the dynamic scheduler. When the dynamic scheduler selects a cluster to be run, the cluster's statically computed schedule is executed.

This kind of behavior is exactly what is provided by Ptolemy's wormhole mechanism, in which a portion of the graph that follows one computational model appears as an atomic actor inside a larger portion of the graph that follows another computational model. Clusters have some of the features of wormholes; for example, **`BDFClusterBag`** has an internal scheduler, and all clusters appear as atomic actors to the outside. However, clusters do not have internal **`Target`** objects, and there are no **`EventHorizon`** objects to convert between **`Particle`** communication protocols because there is no difference in protocol. Accordingly, clusters are designed to serve as "lightweight wormholes" — in particular, cluster boundaries are treated exactly like wormhole boundaries by all schedulers. Given this behavior, all that is necessary to arrange for mixed static and dynamic scheduling is to arrange for the dynamic scheduler to run the galaxy containing the top level clusters, and to assure that the clusters, when run, obey the protocol expected of dynamic actors by the dynamic dataflow scheduler.

4.6. BDF CODE GENERATION FOR A SINGLE PROCESSOR

We now discuss the modifications to the Ptolemy code generation scheme described earlier to permit BDF code generation for single-processor targets. The design goal was to permit all targets to use dynamic actors, not to require that special new targets or new domains be provided. Accordingly, **`CGStar`**, the base class for all code generation stars, is now derived from **`DynDFStar`**, and **`CGPortHole`** is derived from **`DynDFPortHole`**. This means that all code generation domains now permit dynamic actors such as SWITCH and SELECT. However, it is not currently possible to generate code corresponding to Ptolemy's dynamic dataflow scheduler or that handles dynamic memory allocation for arcs; therefore, systems of code generation stars that cannot be completely clustered are rejected as errors.

4.6.1 Additional Methods for Code Generation Targets

So that code can be generated corresponding to the structure built up by the BDF loop scheduler, **`CGTarget`** and derived **`Target`** classes were given new methods that generate the correct code for if-then-else constructs and do-while constructs. There are five new methods, as follows. Separate implementations of these methods must be supplied for each output language targeted.

```
void beginIf(PortHole& cond,int truthdir,
        int depth,int haveElse);
```

This method begins an "if-then" or "if-then-else" statement. Subsequent code generation corresponds to code to be executed if the condition *cond*'s value matches the "truth direction" *truthdir*. The *depth* parameter indicates the nesting depth; if *haveElse* is TRUE, there is an "else" part to the statement.

```
void beginElse(int depth);
```

This method begins the "else" part of an "if-then-else" statement that has previously been begun with a **beginIf** call. The *depth* parameter will match that of the previous **beginIf** call that corresponds to this "else" part. Subsequent code generation corresponds to code that belongs in the "else" part of the statement.

```
void endIf(int depth);
```

This method completes the "if-then-else."

```
void beginDoWhile(int depth);
```

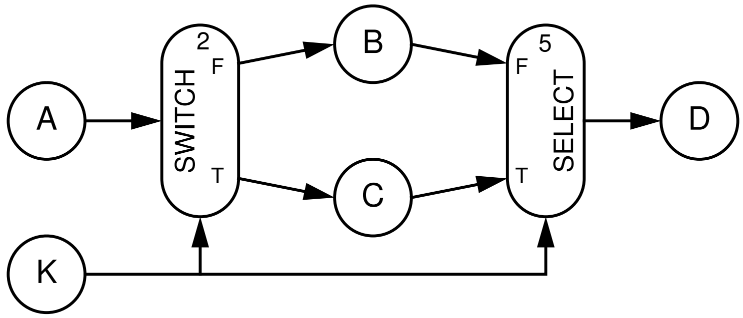
This method begins a "do-while" statement. The condition is provided at the end.

```
void endDoWhile(PortHole& cond,int truthdir,
        int depth);
```

This method ends the "do-while" statement. The loop will continue to execute as long as the state of the condition *cond* matches the truth direction specified by *truthdir.*

4.6.2 Efficient Code Generation for SWITCH and SELECT

It would be possible to generate code for SWITCH and SELECT actors that reads the control token and, based on its value, copies a token between the appropriate pair of arcs. We can do much better, though. Consider the special case in which all ports connected to the SWITCH or SELECT actor transfer only one token. This will be true if all actors adjacent to the SWITCH or SELECT are homogeneous, for example. In this case, all arcs connected to the actor except for the control arc can share the same memory and no code is required to implement the SWITCH or SELECT function. The token on the control arc will still be used; it will be referred to by the control construct that implements the "if-then-else" or "do-while" statement. For example, in the canonical if-then-else construct below



where all actors other than SWITCH and SELECT are homogeneous, we can allocate a single memory location for the value produced by A, and a single memory location for the value consumed by D, and arrange to have the actors B and C share these locations, which is feasible because only one of the two actors will execute. The token generated by actor K determines which of B or C will execute.

In order to have all the arcs share the same buffer, we require that the data input(s) and output(s) of the SWITCH and SELECT be of size one. The current implementation also requires that the control arc have only one token, so that it will be a simple matter to find the control token that controls execution. These restrictions would appear to be a severe limitation, but in practice they are easily met: if a non-homogeneous actor is connected to a SWITCH or a SELECT, the system simply inserts a dummy homogeneous actor that performs a copy operation in between.

It would be possible to remove some of the restrictions on dynamic actors. Consider the SWITCH actor, and assume that one or more of the data arcs transfer more than one token per execution. We can still use one buffer for all three arcs; this would be accomplished by having the actors that read from the TRUE output and the FALSE output of the switch share a read pointer. Since the star connected to the TRUE output is not executed unless the control token is TRUE, and similarly for the star connected to the FALSE output, sharing the read pointer assures that the data are properly "consumed" by the star they are intended for.

The data input and the two outputs of the SWITCH, as well as the data inputs and the output of the SELECT share memory by use of the Ptolemy embedded buffer mechanism, which is described in detail in [Pin93]. The control input to each actor has its own buffer.

4.7. EXAMPLE APPLICATION: TIMING RECOVERY IN A MODEM

This section will consider a nontrivial application of BDF scheduling of a dataflow graph: timing recovery in a modem. The application models baud-rate timing recovery in a digital communication system using an approximate minimum mean square error technique. This example, as implemented in Ptolemy, was presented in [Buc91a]; the underlying digital communication theory is presented in detail in [Lee88b]. The example appears in figure 4.9.

An amplitude-shift keyed (ASK) signal is generated by the "ask" galaxy on the left. The bit source provides a source of random bits; the table lookup actor and pulse

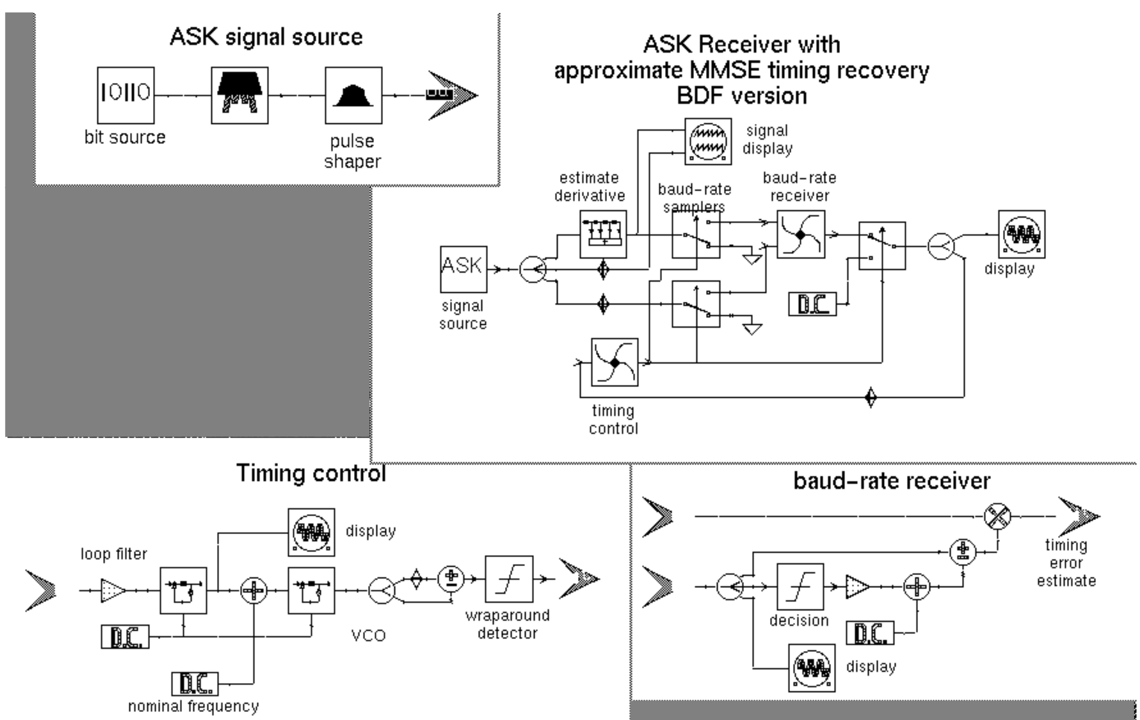


Figure 4.9 A Ptolemy screen dump of an application of BDF graphs to timing recovery in a modem. The top-level system is at the upper right; the other three windows represent subsystems (galaxies).

shaper provide for modulation, resulting in a simple baseband, binary-antipodal signal with a 100% excess bandwidth raised cosine pulse. The sample rate is eight times the baud rate, and may be controlled by adjusting the parameters of the pulse shaping filter. The derivative of the signal is estimated using a finite impulse response (FIR) filter in the top-level diagram (the universe). The derivative and the signal sample itself are sampled by a signal provided by the "timing control" subsystem; they will either be discarded (at convergence, about seven out of eight times) or passed on to the baud rate subsystem (about one out of eight times) by a pair of SWITCH actors. This baud rate subsystem estimates the timing error and uses this estimate to control a phase locked loop. The key to estimating the error is that, if the timing is correct, we should see full-scale values (plus or minus one) at the decision device (the "slicer") and a slope (derivative) of zero. Accordingly, the error estimate is formed by multiplying the estimated derivative by the error at the slicer.

The error estimate is upsampled to the original sample rate at the SELECT actor by adding zeros corresponding to the missing points. It is then used to adjust a phased locked loop implemented in the "Timing Control" galaxy. A simple voltage controlled oscillator is made using an integrator with limits that is allowed to wrap around when the limits are exceeded. The wrap-around is detected and used as the signal to indicate that a baud-rate sample should be taken. Increasing the input to the VCO integrator (middle of the lower left window) causes the time between samples to decrease.

Executing the simulation generates four plots, corresponding to the four graph stars. These plots appear in figure 4.10. The first plot shows the line signal and its estimated derivative. The second and third plots show the timing control signal and the error signal used by the phase locked loop, respectively. The final signal shows the actual samples, representing the received digital data. Ideally the values of these samples will be 1 and −1.

In [Buc91a], the simulation of this system under Ptolemy's DDF and SDF domains was described. Here the three subsystems were statically scheduled and the top-level sys-

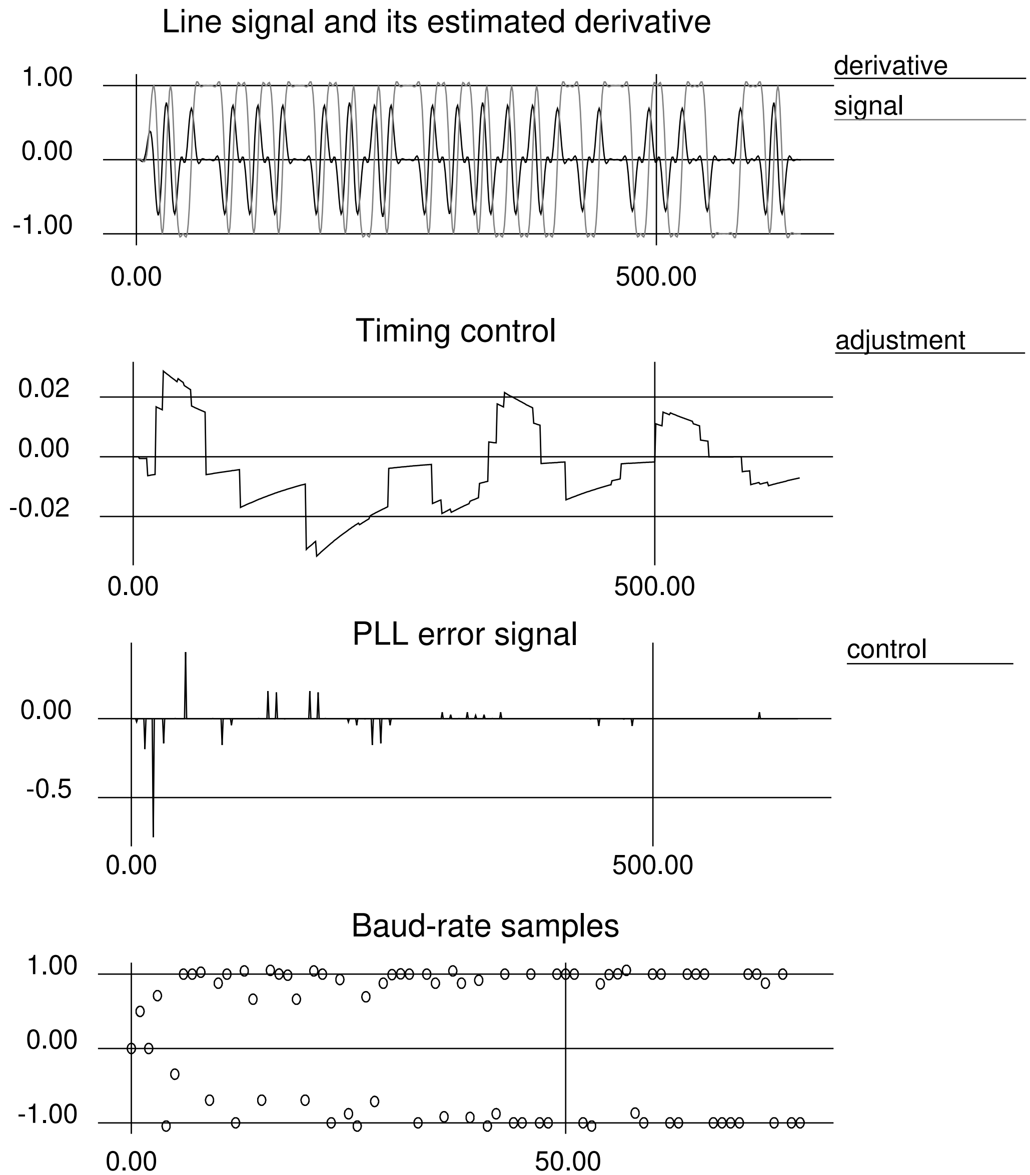


Figure 4.10 Plots generated by the Ptolemy timing recovery model of figure 4.9. The plots show the first 80 baud-rate samples. The sample clock is eight times the baud rate, hence the first three plots have eight times as many samples as the last plot.

tem was dynamically scheduled. However, given a BDF implementation, it is possible to cluster the graph to find an efficient set of control structures that permit a lower-overhead simulation, or to generate code for an even more efficient execution, either in C or in assembly language.

We now describe the clustering of the graph by the BDF loop scheduler. For simulation purposes, it is possible to declare that the subsystems are regular (SDF) as was done in [Buc91a], but it turns out to be more efficient to use BDF scheduling at all levels to avoid the excess overhead of communicating across wormhole boundaries. This is because the BDF loop scheduler generates a static schedule for all regular subsystems found in the graph.

The control structure of the graph is not extremely complex; there is a sample rate change, because the ASK subsystem produces eight samples per execution and the FORK actor consumes one, and there is an if-then-else construct formed by the pair of SWITCH actors and the SELECT. Furthermore, the presence of the four delay tokens complicates the analysis somewhat, though for the most part, these complications come into play only for code generation, since they affect the buffer allocation for arcs.

The system has thirty-six actors, including four implicit FORK actors inserted to permit the same actor output to connect to multiple inputs. The first merge pass succeeds in reducing the universe to seven clusters. This clustering is shown in figure 4.11. Most merging is accomplished by the “fast merge pass” using only local information; to com-

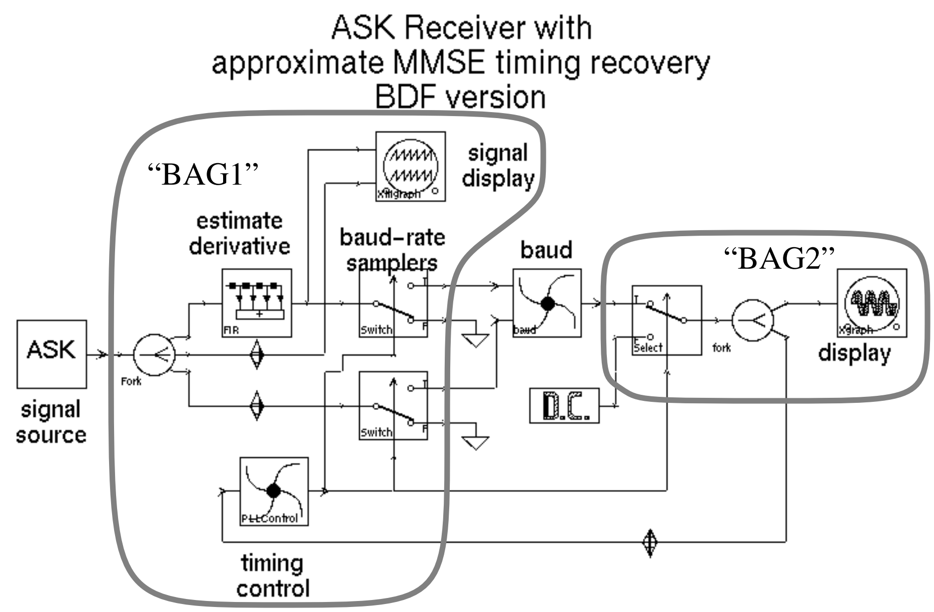


Figure 4.11 Clustering caused by applying the first merge pass. Clusters are indicated by the two loops marked BAG1 and BAG2; also, the subsystems “ASK” and “baud” become single clusters, as do the two “black holes” and the DC star.

bine the two SWITCH actors into the “BAG1” cluster, it is necessary to remap the control arcs for the “baud” and the “black hole” actors (the latter actors are the inverted triangles attached to the FALSE outputs of the SWITCH actors). These arcs are controlled after clustering by the arc that connects to the control input of the SELECT actor that is part of the cluster “BAG2.” Although they have the same repetition rate, BAG1 and BAG2 cannot be combined into one because this would cause deadlock.

The first loop pass makes the two black hole actors, the “baud” cluster/subsystem, and the DC actor into conditionally executed clusters. To do so, a dummy arc is created connecting these clusters with the BAG1 cluster; this arc provides a copy of the control signal. BAG1 is not looped to match the rate of the “ASK” subsystem because of the need to “loop” the “baud” subsystem first. After the loop pass, the next merge pass is able to combine BAG1, BAG2, and all the conditional subsystems into one. There are now only two clusters: the ASK cluster/subsystem and everything else. A “repeat 8 times” loop is put around the “everything else” cluster, and the system has now been completely clustered. At this stage, each of the black hole actors, the DC actor, and the “baud” subsystem is in a different “if” statement; the parallel loop merge pass combines these into a single if-then-else statement. Here is a simplified version of the generated schedule, in which subsystems are written as single actors and automatically inserted forks are omitted:

```
ASK;
repeat 8 times {
      Fork3,TimingControl,FIR1, XMgraph;
      Switch1, Switch2;
      if (TimingControl.output) Baud;
      else { BlackHole1, BlackHole2, DC }
      Select1; Fork; Xgraph3
}
```

When code is generated for this system, no code is required to implement the FORK, SWITCH, and SELECT actors. However, because of initial delay tokens, it turns out that one of the SWITCH actors and the SELECT actor are connected to buffers that require two tokens, violating the assumption used to implement these actors with no code and with embedded buffers. This problem is solved by automatically inserting a pair of COPY actors, whose function is to generate code to copy a single token. Insertion of these extra actors implies the creation of two extra buffers. In effect, we have added a small amount of code to create two simple delay lines.

4.8. SUMMARY AND STATUS

At the present time, Ptolemy's ability to execute BDF actors in a simulation mode is nearly complete. Dataflow graphs with mixtures of BDF and SDF (regular) actors are clustered as much as possible, and the clusters are dynamically executed if the algorithm does not successfully reduce the graph to a single cluster. Other than the special case of determining that do-while loops are valid, the state traversal algorithm described in section 3.4.1 is not implemented.

Code generation using the BDF model is currently limited to C language generation for a single processor, and assembly language BDF code generation will be completed shortly.

5

EXTENDING THE BDF MODEL

God made the integers; all else is the work of man.

— Kronecker

This chapter describes an extension of the token flow model that permits a larger class of dynamic dataflow actors to be considered. This class differs from BDF actors such as SWITCH and SELECT in that control tokens are permitted to have arbitrary integer values, not just TRUE and FALSE. We will find that, for the most part, the analysis techniques developed in previous chapters apply with little change to this extended model, which we shall call integer-controlled dataflow, or IDF.

5.1. MOTIVATION FOR INTEGER-VALUED CONTROL TOKENS

While the Boolean-controlled dataflow model is Turing-equivalent, it does not directly express certain actors that have been found to be useful. Most of these actors have the property that the control token is an integer rather than a Boolean token, which might be used in two ways:

- Specification of the number of tokens produced or consumed on some arc (e.g. the REPEAT or LAST_OF_N actor), or
- Enabling or disabling the arc depending on whether the token has a specific value or belongs to some set of values (as in a multi-way CASE construct).

Note that it is not difficult to synthesize either a REPEAT actor or a multi-way CASE from the SWITCH, SELECT, and SDF actors. In some cases, however, the constructs that naturally arise for iterations have shortcomings. Consider the design of a subgraph that, given an integer-valued token with value *n*, computes a token with value

$$g(n) = \sum_{i=0}^{n-1} f(i) \tag{4-1}$$

assuming that the function $f(n)$ is computed by an atomic actor. Let us assume that the function *f* is relatively expensive to evaluate, and we wish to leave open the possi-

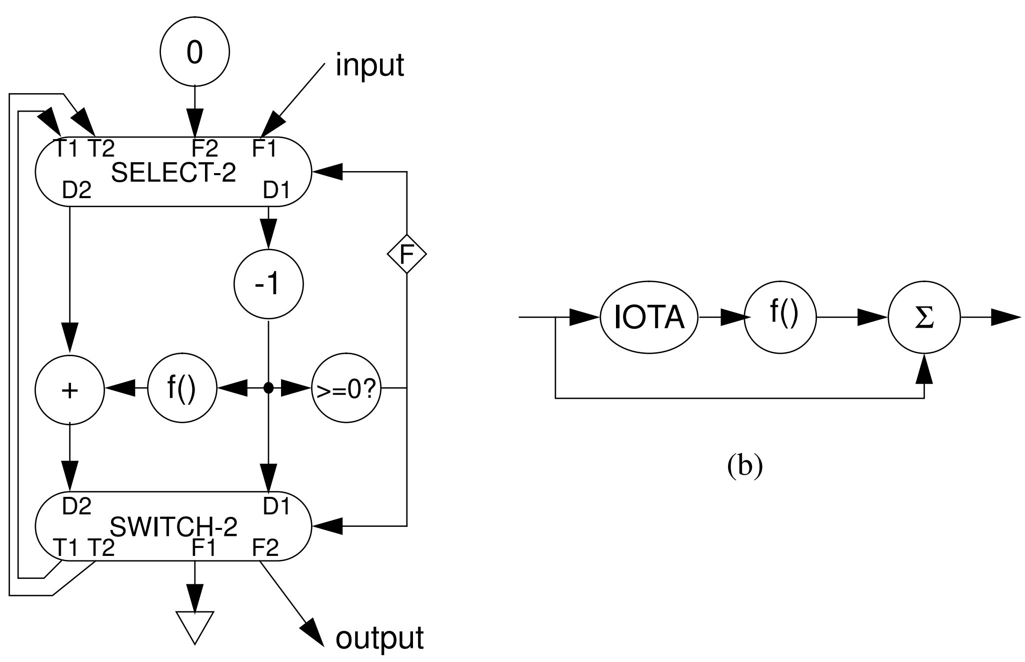


Figure 5.1 The first subgraph (a) implements the function $g(n)$ described above using BDF actors. The actors SWITCH-2 and SELECT-2 switch two data streams based on one control token, e.g. SWITCH-2 copies D1 to either T1 or F1 and copies D2 to either T2 or F2. The system on the right (b) computes the same function using coarser-grained IDF actors.

bility that the f evaluations be computed in parallel. We could produce a subgraph that implements this function using BDF and a DO-WHILE loop (see figure 5.1 (a)), but this graph implies a serial execution of the f actors, and the data dependency between the iterations is difficult to analyze away. The parallelism is more naturally expressed with actors that have integer control tokens. Consider two such actors: one that, given an integer value n, produces n output tokens with values ranging from 0 to $n-1$, and one that, given an integer value n on its control port, reads n tokens from its input data port and outputs their sum. Let us call the former actor IOTA (after the operation from the APL language that it resembles) and the latter actor SUM or Σ. Then the simple system in figure 5.1 (b) naturally models the solution. While it is true that we could produce BDF systems corresponding to the actors IOTA and SUM, it would be desirable to have a theory that could represent such actors directly, rather than as composite systems of simpler actors. However, the BDF model has one very significant advantage: the BDF system requires only one location for each arc, while the IDF system requires memory proportional to n.

We therefore wish to extend the BDF model to permit integer control tokens. We will consider a set of actors with the following properties: the number of tokens produced or consumed on any arc is either a constant, or a function of an integer-valued control token on some other arc of the same actor. Only the following functions are permitted:

Type 1 (CASE): the number of tokens transferred is either a constant, or zero, depending on whether the value of the control token is a member of a specified set.

Type 2 (REPEAT): the number of tokens transferred is a constant multiple of the control token.

Given any specified encoding of TRUE and FALSE values into integers, we see that BDF actors are IDF actors. If only Type 1 functions are considered, there is not much new in the IDF theory: we simply have mapping functions to turn integer tokens into Booleans, and, with respect to any controlled arc, a control token may still be regarded as “true” or “false”. However, relations among Booleans may be more easily discovered and

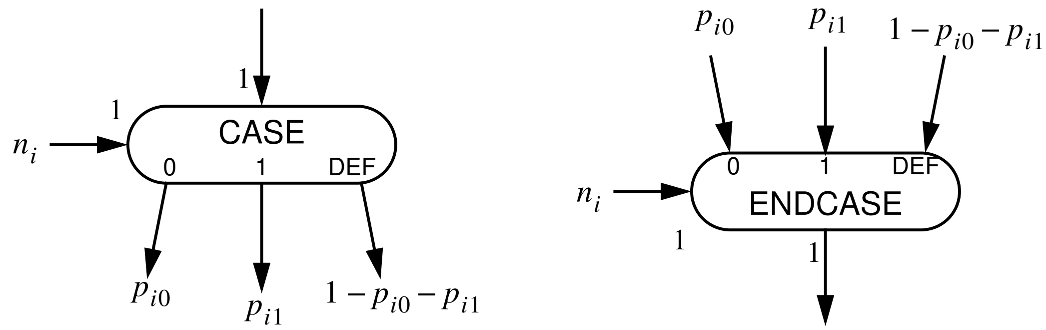


Figure 5.2 The CASE and ENDCASE actors, annotated with IDF analysis quantities. This particular pair of actors implement a three-way case, however, any number of outputs are admissible.

represented in some cases given CASE arcs.

We introduce two new dynamic dataflow actors, which we call CASE and ENDCASE, as shown in figure 5.2. CASE is the IDF analog of the SWITCH actor, and ENDCASE is the IDF analog of the SELECT actor.

5.2. ANALYSIS OF IDF GRAPHS

Where in BDF theory we use p_n to represent the proportion of tokens on stream n that are TRUE, we instead use p_{nm} to represent the proportion of tokens on stream n that have value m. The other interpretations for the p quantities we considered for the BDF case, such as long-term averages and probabilities, could be used as well for the IDF case, of course. The analysis problems are much the same as before: the procedure for determining whether a graph has a bounded length schedule is the same as before, and the clustering algorithm is easily adapted to handle “CASE arcs”. The result is that we now build multi-way case statements where in BDF we built if-then-else statements, so we have generalized from a two-way conditional branch to an N-way conditional branch.

Using the CASE and ENDCASE actors, we can produce the three-way branch analog to the canonical if-then-else construct, as shown in figure 5.3. By a straightforward generalization of the techniques of section 3.2, we can determine the repetition vector for the graph; it is simply

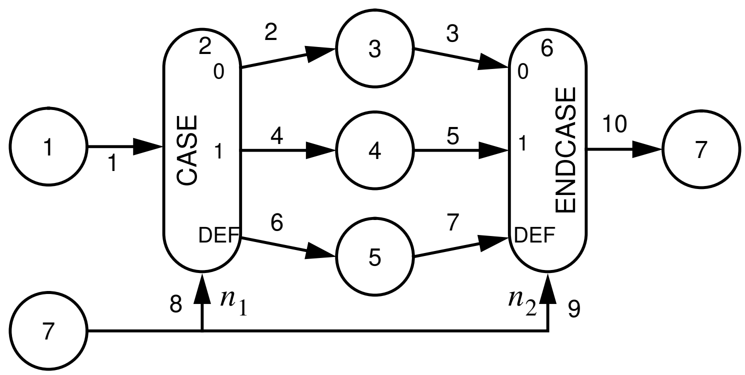


Figure 5.3 A three-way CASE statement. The numbers adjacent to arcs and on actors merely identify them; all actors other than CASE and ENDCASE are homogeneous.

$$\vec{r}(\vec{p}) = k\begin{bmatrix}1 & 1 & p_{10} & p_{11} & (1-p_{10}-p_{11}) & 1 & 1 & 1\end{bmatrix}^T \tag{4-2}$$

By analogy with BDF theory, we now interpret expressions like p_{10} as the number of tokens on control stream 1 with value 0 during a complete cycle divided by the number of tokens on stream 1 in a complete cycle, and then find the smallest integer solution. We then find that there is only one control token per complete cycle and the repetition vector is

$$\vec{r}(\vec{p}) = \begin{bmatrix}1 & 1 & n_0 & n_1 & (1-n_0-n_1) & 1 & 1 & 1\end{bmatrix}^T \tag{4-3}$$

where n_0 is 1 if the control token is 0 and 0 otherwise, and n_1 is 1 if the control token is 1 and 0 otherwise. Furthermore, it is a simple matter to generalize the clustering algorithm of section 3.3 to cluster graphs such as this to form multi-way case statements, like the "switch" construct in the C programming language.

Type 2 arcs, in which the number of tokens transferred on an arc is proportional to the value of an integer control token, introduce a new complication into IDF theory. If we have even a single type 2 arc in the system, we immediately have unbounded memory, because there is no limit on how large an integer control token's value might be[1]. But

1. For certain actors, it might be possible to exploit properties of the actor's semantics to avoid unbounded memory. For example, all outputs of a REPEAT actor have the same value, and depending on the context, it might be possible to use a size-1 shared buffer to hold the value rather than a buffer of unbounded size.

there are very distinct differences between a case like the IDF graph of figure 5.1 (b) and a BDF graph with data-dependent iteration. The BDF graph may represent a system that never halts; however, we can be assured that the IDF system always terminates. With the IDF system, it is also a simpler matter to determine the number of times each actor is executed. While the cycle length and the memory required are not absolutely bounded, both are bounded if we possess an upper bound on the value of the computed tokens, and furthermore they are guaranteed to be finite even without such a bound. Thus for IDF we have an important distinction between “bounded length schedule” and “finite length schedule” and we can speak of bounds that are functions of the maximum values of certain control tokens.

It may be possible to combine the advantages of IDF and BDF in cases like figure 5.1. Note that we could construct subsystems with behavior corresponding to the IOTA and SUM actors of figure 5.1 (b) out of BDF actors. IDF analysis permits us to easily determine the number of executions of each actor. We can now remove the cluster boundaries of the IOTA and SUM systems and schedule the collection of actors as BDF actors, thereby assuring that memory is bounded. What we have accomplished that could not be obtained by BDF theory alone is that we know the number of times the actors are executed; since BDF knows only about relationships between Boolean tokens and knows nothing about the properties of the DECREMENT and COMPARE-TO-ZERO actors that might make up IOTA and SUM, it is not capable of reaching these conclusions.

6

FURTHER WORK

Graduate? But I'm not done yet!

— J. T. Buck

It is rare indeed when any line of research can be considered completed, and there is much remaining work to do on the token flow model. This chapter summarizes the principal lines of investigation now being considered as an answer to the question "What's next?". There are theoretical issues having to do with answering open questions about the material presented in Chapter 3, implementation of the current theory is incomplete, and there is also the task of extending BDF to fully support parallel scheduling. The last topic, parallel scheduling of dynamic dataflow graphs, is worthy of a thesis topic all on its own, and [Ha92] is such a thesis, and has a bibliography pointing to other work in the field. To avoid significantly expanding the size of this thesis for little gain, we will not attempt to duplicate the full treatment of the topic given there, but rather we will simply summarize possible approaches to the use of BDF theory for parallel scheduling.

6.1. IMPROVING THE CLUSTERING ALGORITHM

There is a striking contrast between the completeness of the development of the loop scheduling theory in [Bha93b] and the clustering algorithm presented in section 3.3. To summarize, we find that we can completely cluster regular dataflow graphs into single appearance schedules provided that they have no tightly interdependent components, and we have algorithms for finding such components. Even if tightly interdependent components exist, we can still find efficient looped schedules for the remainder of the graph, with repeated appearances only for the actors that appear in the tightly interdependent components. A family of divide-and-conquer algorithms is presented that decomposes the loop scheduling problem for regular dataflow graphs into a set of smaller problems.

No such complete theory exists for the BDF loop scheduling problem; instead, we have presented a series of transformations that simplify the structure but that may not succeed in completely clustering it, without any sort of precise indication of the properties of the class of graphs that can be completely clustered. One possible line of investigation is to find divide-and-conquer algorithms for BDF graphs that attempt to separate out the parts of the graph whose execution depends on particular Boolean streams.

6.2. PROVING THAT UNBOUNDED MEMORY IS REQUIRED

In section 3.4.2, we provide a technique for proving that a BDF graph requires unbounded memory. It appears that the fourth condition we give for proving that unbounded memory is required, given that we can reach state μ' from state μ, is too strong. Is it really required that we check every intermediate state as described on page 98? This check may be expensive in some cases, and may not even be needed given that the first three conditions are satisfied. At minimum, it should be possible to find a weaker condition.

6.3. USE OF ASSERTIONS

We have at several points discussed the use of user-supplied assertions to provide the BDF analysis system with more information than it can directly obtain from the graph.

One assertion that is relatively easy to use is the statement that two Boolean streams are equal. Such assertions can be added when a system is found to be only weakly consistent because the system could not prove two streams to be equal, as in the example in figure 4.6, as discussed in section 4.3.1. Given this type of assertion, the clustering algorithm can usually reduce the graph to standard control structures.

There is another type of assertion that may be useful in cases where the state space would otherwise be unbounded, for example in figure 3.5. If we knew, for example, that the Boolean control stream in this actor could never contain more than 10 TRUE tokens in a row, then the graph could be scheduled in bounded memory. The state space in such cases could be quite large, and it would be desirable to find efficient ways to handle such cases in an efficient way.

6.4. PARALLEL SCHEDULING OF BDF GRAPHS

Parallel scheduling of dynamic dataflow graphs is a topic worthy of a dissertation in itself; in fact, my colleague Soonhoi Ha recently completed such a project [Ha92]. It has not, however, been the main focus of research on the token flow model, which has been concerned mainly with the consistency properties of the graphs and with the generation of sequential schedules. However, it has always been our intention to extend the analysis principles of the token flow model to encompass parallel scheduling, and accordingly this section points out directions for parallel scheduling of such graphs. This section is unavoidably sketchy and lacking in detail.

The first possibility is to build on the work of Lee [Lee88] and Ha [Ha91], [Ha92], in which standard dynamic constructs are scheduled using quasi-static methods. These techniques produce parallel schedules based on the simplifying assumption that the control stream that controls each dynamic construct (if-then-else, multi-way switch, do-while, or recursion) has known statistics and that these streams can be considered as being independent of each other. These assumptions are clearly violated in practice, but at least yield a good starting point. By coupling the BDF clustering algorithm (and its generalization to

IDF) with this quasi-static scheduling framework, a greater variety of control structures could be automatically recognized.

For the case of BDF graphs with bounded-length schedules, another approach is feasible that does not require any assumptions about the statistics of the Boolean control streams. In this approach, which is appropriate for hard real-time systems in which deadlines must be met, our scheduling criterion is to minimize the worst-case execution time of the schedule, or to produce a schedule that assures that a deadline is met regardless of the outcomes of the Boolean control streams. For all but the most trivial cases, either of these criteria lead to an NP-complete problem, meaning that it belongs to a class of problems for which the only known solutions require time that is exponential in the size of the problem (see [Gar79] for a thorough discussion on the theory of NP-completeness). We must therefore settle for heuristic, suboptimal solutions.

It appears reasonable to generalize heuristic algorithms based on the critical path algorithm [Ada74], which generates near-optimal schedules when communication costs are not included, or the various heuristic algorithms discussed in [Sih91] that do take communication costs into account, to work with the annotated acyclic precedence graphs discussed in section 3.2.2. In effect, we generate one schedule for each of the possible Boolean outcomes. There are some complications added by the requirement that Boolean control tokens be communicated between processors when a computation on one processor depends on Boolean tokens generated on another processor. This may effect the communication costs generated by certain partitions.

REFERENCES

[Ack79]

W. B. Ackerman and J.B. Dennis, “VAL — A Value-oriented Algorithmic Language; Preliminary Reference Manual,” Laboratory for Computer Science MIT/LCS/TR-218, MIT, 1979.

[Ada74]

T. L. Adam, K. M. Chandy, and J. R. Dickson, “A Comparison of List Schedules for Parallel Processing Systems,” *Communications of the ACM*, **17(12)**, pp. 685-690, December 1974.

[Aga93]

A. Agarwal, J. Kubiatowicz *et al*., “Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors,” *IEEE Micro*, June 1993.

[Aho86]

A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[All72]

F. E. Allen and John Cocke, “Graph-Theoretic Constructs for Program Flow Analysis,” IBM Technical Report RC 3923 (17789), IBM Thomas J. Watson Research Center, July 1972.

[Amb92]

A. L. Ambler, M. M. Burnet, B. A. Zimmerman, “Operational Versus Definitional: A Perspective on Programming Paradigms,” *Computer*, September 1992.

[Arv80]

Arvind and R. E. Thomas, “I-Structures: An Efficient Data Type For Functional Languages,” Technical Report LCS/TM-178, MIT, 1980.

[Arv82]

Arvind and K. P. Gostelow, “The U-Interpreter,” *Computer*, **15(2)**, February

1982.

[Arv86]

Arvind and D. E. Culler, “Dataflow Architectures,” *Annual Review in Computer Science*, **Vol. 1**, pp. 225-253, 1986.

[Arv90]

Arvind and R. S. Nikhil, “Executing a Program on the MIT Tagged-Token Dataflow Architecture,” *IEEE Trans. on Computers*, **39(3)**, pp. 300-318, March 1990.

[Arv91]

Arvind, L. Bic, and T. Ungerer, “Evolution of Data-Flow Computers,” in *Advanced Topics in Dataflow Computing*, ed. L. Bic and J.-L. Gaudiot, Prentice Hall, 1991

[Ash75]

E. A. Ashcroft, “Proving Assertions about Parallel Programs,” *J. of Computer and Systems Science*, **10(1)**, pp. 110-135, 1975.

[Ash77]

E. A. Ashcroft and W. W. Wadge, “Lucid, a Nonprocedural Language with Iteration,” *Comm. ACM,* **20(7)**, pp. 519-526, July 1977.

[Bac78]

J. Backus, “Can programming be liberated from the von Neumann style? A functional style and its algebra of programs,” *Communications of the ACM*, **21(8)**, pp 613-641, 1978.

[Bal89]

H. E. Bal, J. G. Steiner, A. S. Tanenbaum, “Programming Languages for Distributed Computing Systems,” *ACM Computing Surveys*, **21(3)**, September 1989, pp. 359-411.

[Bal90]

R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein, “The Program Dependence

Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages," *Proc. of the ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pp. 257-271, June 1990.

[Ben90]

A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Trans. on Automatic Control,* **35(5)**, pp. 535-546, May 1990.

[Ber92]

G. Berry and G. Gonthier, "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, **19(2)**, pp. 87-152, Nov. 1992.

[Ber93]

G. Berry, S. Ramesh, and R. K. Shyamasundar, "Communicating Reactive Processes," *Proc. 20th ACM Conf. on Principles of Programming Languages*, Charleston, VA, 1993.

[Bha91]

S. S. Bhattacharyya, "Scheduling Synchronous Dataflow Graphs for Efficient Iteration," Master's Thesis, EECS Dept. Univ. of Calif. Berkeley, May, 1991.

[Bha93a]

S. S. Bhattacharyya and E. A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," **to appear** in *J. of VLSI Signal Processing,* 1993.

[Bha93b]

S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, "A Compiler Scheduling Framework for Minimizing Memory Requirements of Multirate DSP Systems Represented as Dataflow Graphs," Memorandum UCB/ERL M93/31, Electronic Research Laboratory, U. California, Berkeley, March 1993.

[Bic91]

L. Bic, M. D. Nagel, and J. M. A. Roy, "On Array Partitioning in PODS," in *Advanced Topics in Dataflow Computing*, ed. L. Bic and J.-L. Gaudiot, Prentice Hall, 1991

[Bie90]

J. Bier, E. Goei, W. Ho, P. Lapsley, M. O'Reilly, G. Sih and E.A. Lee, "Gabriel: A Design Environment for DSP," *IEEE Micro Magazine*, **10(5)**, pp. 28-45, October 1990.

[Böh91]

A. P. W. Böhm, R. R. Oldehoeft, D. C. Cann, J. T. Feo, "SISAL 2.0 Reference Manual," Colorado State University Computer Science Department Technical Report CS-91-118, November 1991.

[Boo89]

G. S. Boolos and R. C. Jeffrey, *Computability and Logic*, Third Edition, Cambridge University Press, 1989.

[Boo91]

G. Booch, *Object Oriented Design With Applications*, Benjamin/Cummings, 1991.

[Buc91a]

J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Multirate Signal Processing in Ptolemy," *Proc. ICASSP 1991*, Toronto, Canada, April 1991.

[Buc91b]

J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: a Platform for Heterogeneous Simulation and Prototyping," *Proc. 1991 European Simulation Conference*, Copenhagen, Denmark, June 1991.

[Buc91c]

J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: a Mixed Paradigm Simulation/Prototyping Platform in C++," *Conference Proceedings, C++ At*

Work 1991, Santa Clara, California, November 1991.

[Buc92]

J. T. Buck and E. A. Lee, “The Token Flow Model,” presented at *Data Flow Workshop*, Hamilton Island, Australia, May, 1992.

[Buc93a]

J. T. Buck and E. A. Lee, “Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model,” *Proc. of ICASSP ‘93*, Minneapolis, MN, April, 1993

[Buc93b]

J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” t**o appear** in *International Journal of Computer Simulation*, special issue on “Simulation Software Development,” 1993

[Buc93c]

J. T. Buck, “The Ptolemy Kernel: A Programmer’s Guide to Ptolemy version 0.4,” Memorandum UCB/ERL M93/8, January 19, 1993.

[Bur92]

M. Burns, “SISAL Challenges Fortran,” *Supercomputing Review,***5(2)** pp. 72-73, February 1992.

[Cas92]

P. Caspi, “Clocks in Dataflow Languages,” *Theoretical Computer Science*, **94(1)**, pp. 125-140, March 1992.

[Chu32]

A. Church, “A set of postulates for the foundation of logic,” *Ann. Math. 2*, 33-34, 346-366, 839-964, 1932.

[Com72]

F. Commoner, “Deadlocks in Petri Nets,” Report CA-7206-2311, Massachusetts

Computer Associates, Wakefield, MA (June 1972), 50 pp.

[Cul89]

D. E. Culler, "Managing Parallelism and Resources in Scientific Dataflow Programs," Ph.D. thesis, MIT, June 1989.

[Cyt89]

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An Efficient Method of Computing Static Single Assignment Form," *Proc. of 16th ACM Symp. on Principles of Programming Languages*, pp. 25-35, January 1989.

[Dav78]

A. L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," *Proc. of the Fifth Annual Symposium on Computer Architecture*, pp. 210-215, ACM, April 1978.

[Den75a]

J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Dataflow Processor," *Proc. 2nd Ann. Symp. Computer Architecture*, New York, May, 1975.

[Den75b]

J. B. Dennis, "First Version Data Flow Procedure Language," Technical Memo MAC TM61, May, 1975, MIT Laboratory for Computer Science.

[Den80]

J. B. Dennis, "Data Flow Supercomputers," *Computer*,**13(11)**, November 1980.

[Den91]

J. B. Dennis, "The Evolution of 'Static' Data-Flow Architecture," in *Advanced Topics in Dataflow Computing*, ed. L. Bic and J.-L. Gaudiot, Prentice Hall, 1991.

[Den78]

P. J. Denning, J. B. Dennis, and J. E. Qualitz, *Machines, Languages, and Computation,* Prentice-Hall, 1978.

[Des93]

D. Desmet and D. Genin, “ASSYNT: Efficient Assembly Code Generation for DSP’s Starting from a Data Flowgraph,” *Proc. ICASSP 1993*, Minneapolis, April, 1993.

[Evr91]

P. Evripidou and J.-L. Gaudiot, “The USC Decoupled Multilevel Data-Flow Execution Model,” in *Advanced Topics in Dataflow Computing*, ed. L. Bic and J.-L. Gaudiot, Prentice Hall, 1991.

[Fer87]

J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The Program Dependence Graph and its Use in Optimization,” *ACM Trans. on Programming Languages and Systems*, **9(3)**, pp. 319-349, July 1987.

[Flo79]

R. Floyd, “The Paradigms of Programming (Turing Award Lecture),” *Comm. ACM,* **22(8)**, pp. 455-460, August 1979.

[Fly72]

M. J. Flynn, “Some Computer Organizations and Their Effectiveness,” *IEEE Trans. on Computers* **C-21(9)**, pp. 938-960, September 1972.

[Gar79]

M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Feeman and Co., New York, NY, 1979.

[Gao88]

G. R. Gao, R. Tio, and H. J. Hum, “Design of an Efficient Dataflow Architecture Without Dataflow,” *Proc. of the International Conf. on Fifth-Generation Computers*, pp. 861-868, Tokyo, Japan, December 1988.

[Gao92]

G. R. Gao, R. Govindarajan, P. Panangaden, “Well-Behaved Dataflow Programs for DSP Computation," *Proc. ICASSP 1992*, San Francisco, CA, March 1992.

[Gel93]

P. R. Gelabert and T. P. Barnwell III, "Optimal automatic periodic multiprocessor scheduler for fully specified flow graphs," *IEEE Trans. on Signal Processing*, **41(2)**, pp. 858-888, February 1993.

[Gra90]

V. G. Grafe and J. E. Hoch, "The EPSILON-2 hybrid dataflow architecture," in *COMPCON Spring 1990 Digest of Papers*, 1990.

[Gur85]

J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Communications of the ACM*, **28(1)**, pp. 34-52, January 1985.

[Ha91]

S. Ha and E. A. Lee, "Compile-time Scheduling and Assignment of Dataflow Program Graphs with Data-Dependent Iteration," *IEEE Trans. on Computers*, **40(11)**, pp. 1225-1238, November 1991.

[Ha92]

S. Ha, "Compile-Time Scheduling of Dataflow Program Graphs With Dynamic Constructs," Memorandum No. UCB/ERL M92/43 (Ph.D. Thesis), University of California, Berkeley, April 1992.

[Hac74]

M. Hack, "Decision Problems for Petri Nets and Vector Addition Systems," Computation Structures Group Memo 95, Project MAC, Massachusetts Institute of Technology, Cambridge, MA, March 1974.

[Hal91]

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proc. of the IEEE*, **79(9)**, September 1991.

[Har87]

D. Harel, "Statecharts: A Visual Approach to Complex Systems," in *Science of Computer Programming*, **8-3**, pp. 231-275, 1987.

[Haw88]

S. W. Hawking, *A Brief History of Time: From the Big Bang to Black Holes,* Bantam Books, New York, 1988.

[Hen90]

J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

[Hoa93]

P. D. Hoang and J. M. Rabaey, "Scheduling of DSP Programs Onto Multiprocessors for Maximum Throughput," *IEEE Trans. on Signal Processing*, 41-6, pp. 2225-2235, June 1993.

[How90]

S. How, "Code Generation for Multirate DSP Systems in Gabriel," MS Report, ERL, EECS Dept., UC Berkeley, May, 1990.

[Hu61]

T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, **9(6)**, pp. 841-848, 1961.

[Hud89]

P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages", *ACM Computing Surveys*, Vol 21, No 3, September 1989.

[Kah74]

G. Kahn, "The Semantics of a Simple Language For Parallel Programming," in *Information Processing 74: Proceedings of IFIP Congress 74*, pp. 471-475, August 1974.

[Kah77]

G. Kahn and D. B. MacQueen, “Coroutines and Networks of Parallel Processes,” *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co, 1977.

[Kal92]

A. Kalavade and E. A. Lee, “Hardware/Software Co-Design Using Ptolemy: A Case Study,” *Proceedings of the IFIP International Workshop on Hardware/Software Co-Design*, Grassau, Germany, May 19-21, 1992.

[Kar66]

R. M. Karp and R. E. Miller, “Properties of a Model for Parallel Computation: Determinacy, Termination, and Queueing,” *SIAM Journal of Applied Math*, **14(6)**, pp 1390-1411, November 1966.

[Kar69]

R. M. Karp and R. E. Miller, “Parallel Programming Schemata,” *Journal of Computer and System Sciences*, **3(2)**, pp. 147-195, May 1969.

[Klu92]

W. Kluge, *The Organization of Reduction, Data Flow, and Control Flow Systems*, Massachusetts Institute of Technology Press, 1992.

[Kos78]

P. R. Kosinski, “A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs,” *Conf. Record of the 5th Ann. ACM Symp. on Principles of Programming Languages*, Tuscon, AZ, 1978.

[Kuh62]

T. Kuhn, “The Structure of Scientific Revolutions”, University of Chicago Press, 1962 (second edition published 1970)

[Lee87a]

E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous DataFlow

Graphs for Digital Signal Processing," *IEEE Trans. on Computers*, January, 1987.

[Lee87b]

E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *IEEE Proceedings*, September, 1987.

[Lee88a]

E. A. Lee, "Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages," in *VLSI Signal Processing III*, IEEE Press, 1988.

[Lee88b]

E. A. Lee and D. G. Messerschmitt, *Digital Communication*, Kluwer Academic Press, Norwood, MA, 1988.

[Lee89]

E. A. Lee and S. Ha, "Scheduling Strategies for Multiprocessor Real-Time DSP," *GLOBECOM*, November, 1989.

[Lee91a]

E. A. Lee, "Static Scheduling of Data-Flow Programs for DSP," in *Advanced Topics in Dataflow Computing*, ed. L. Bic and J.-L. Gaudiot, Prentice Hall, 1991

[Lee91b]

E. A. Lee, "Consistency in Dataflow Graphs," *IEEE Trans. on Parallel and Distributed Systems*, **Vol. 2**, **No. 2**, April, 1991.

[LeG91]

P. Le Guernic and T. Gautier, "Data-Flow to von Neumann: the SIGNAL Approach," in *Advanced Topics in Dataflow Computing*, ed. L. Bic and J.-L. Gaudiot, Prentice Hall, 1991.

[Mal87]

J. Malpas, *Prolog: A Relational Language and its Applications*, Prentice Hall, 1987.

[McG83]

J. McGraw, S. Allan, J. Glauert, and I. Dobes, “SISAL: Streams and Iteration in a Single-Assignment Language: Language Reference Manual.” Tech Rep. M-146, Lawrence Livermore National Laboratory, 1983.

[Mes84]

D. G. Messerschmitt, “A Tool for Structured Functional Simulation,” *IEEE Journal on Selected Areas in Communications*, **SAC-2(1)**, January, 1984.

[Mur89]

T. Murata, “Petri Nets: Properties, Analysis, and Applications,” *Proceedings of the IEEE*, **77(4)**, April 1989.

[Nik86]

R. S. Nikhil, K. Pingali, and Arvind, “Id Nouveau,” Computation Structures Group Memo 265, Laboratory for Computer Science, Massachusetts Institute of Technology.

[Pap88]

G. M. Papadopoulos, “Implementation of a General Purpose Dataflow Multiprocessor,” Technical Report TR-432, MIT Laboratory for Computer Science, Cambridge, MA, August, 1988.

[Pet81]

J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Pin85]

K. Pingali and Arvind, “Efficient Demand-Driven Evaluation. Part I,” *ACM Trans. on Programming Languages and Systems,* **7(2)**, pp. 311-333, April 1985.

[Pin93]

J. Pino, S. Ha, E. Lee, and J. Buck, “Software Synthesis for DSP Using Ptolemy,” invited paper in *Journal of VLSI Signal Processing*, to appear.

[Pow92]

D. G. Powell, E. A. Lee, and W. C. Newman, “Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams,” *Proc. ICASSP 1992*, San Francisco, vol. 5, pp. 553-556.

[Rab91]

J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, “Fast Prototyping of Datapath-Intensive Architectures,” *IEEE Design and Test of Computers,* **8(2)**, 1991, p. 40-51.

[Rit93]

S. Ritz, M. Pankert, and H. Meyz, “Optimum Vectorization of Scalable Synchronous Dataflow Graphs”, Technical Report IS2/DSP93.1a, Aachen University of Technology, Germany, January 1993.

[Sat92]

M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, *et al*, “Thread-based Programming for the EM-4 Hybrid Dataflow Machine,” *Computer Architecture News*, **20(2)**, pp. 146-155, May 1992.

[Sch86]

D. A. Schwartz and T. P. Barnwell III, “Cyclo-Static Solutions: Optimal Multiprocessor Realizations of Recursive Algorithms,” in *VLSI Signal Processing*, IEEE Press, 1986.

[Sih91]

G. C. Sih, “Multiprocessor Scheduling to Account for Interprocessor Communication,” Memorandum No. UCB/ERL M91/29 (Ph.D. Thesis), U. C. Berkeley, 1991.

[Ski91]

D. Skillicorn, “Stream Languages and Data-Flow,” in *Advanced Topics in Dataflow Computing*, ed. L. Bic and J.-L. Gaudiot, Prentice Hall, 1991.

[Sto88]

P. D. Stotts, “The PFG Language: Visual Programming for Concurrent Computing,” *Proc. Int. Conf. on Parallel Programming,* Vol. 2, pp. 72-79, 1988.

[Tra91]

K. R. Traub, “Multi-Thread Code Generation for Dataflow Architectures From Non-Strict Programs,” in *Functional Languages and Computer Architecture, 5th ACM Conference Proceedings*, ed. J. Hughes, Springer-Verlag, pp. 73-101, 1991.

[Tur81]

D. A. Turner, “The Semantic Elegance of Applicative Languages,” *Proc. of the ACM Conf. on Functional Programming Languages and Computer Architecture*, Portsmouth, NH, pp. 85-92, 1981.

[Wen75]

K.-S. Weng, “Stream-Oriented Computation in Recursive Data Flow Schemas,” Laboratory for Computer Science (TM-68), MIT, Cambridge, MA, Oct. 1975.

[Whi92]

G. S. Whitcomb and A. R. Newton, “Data-Flow/Event Graphs,” Memorandum No. UCB/ERL M92/24, Electronics Research Lab, University of California, Berkeley, March 4, 1992.