Specification and Verification of Real-Time Constraints in Coarse-Grain Dataflow

Dana S. Henry

MIT / LCS / TR-487
May 1991

This report describes research done at Schlumberger Corporation and written at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

This report was originally published as the author's Master's thesis.

Specification and Verification of Real-Time Constraints in Coarse-Grain Dataflow

Dana S. Henry

Technical Report MIT / LCS / TR-487
May 1991

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

We present a method for verifying real-time constraints in a distributed, coarse-grain dataflow environment starting with a program which has already been allocated onto a machine. The user specifies the timing of each module together with real-time constraints; and we verify the constraints. To deduce program's timing, the user specifies all possible behaviors of each dataflow module and assigns timing costs to each module's behavior. We use the behavior and timing of individual modules to derive a data independent timing model for the entire program. User specifiable constraints include conditional constraints and constraints through non-deterministic paths. An event-driven verification verifies constraints. We justify the need for an event-driven verification, describe design issues, and offer a tagging scheme for sharing state among multiple verifications.

Key Words and Phrases: hard real-time, deadline, specification, verification.

Acknowledgements

Many thanks to every one who has helped in the preparation of this thesis. Special thanks to Professor Arvind, my MIT advisor, for his genuine interest, invaluable support, and precious advise which guided the structure and emphasis of this work. Special thanks also to Dr. David Barstow, my Schlumberger advisor, for his continuing encouragement, his invaluable trust, and the wonderful research environment he has provided. Thanks to Schlumberger-Doll Lab for Computer Science and its members for providing a conducive environment during the time I spent working there on this thesis. Thanks to members of the MIT LCS Computation Structures Group for providing a friendly atmosphere at all other times. And, finally, endless thanks and gratitude to my mother and my grandmother for their unconditional love and support always.

Contents

Chapter 1

Introduction

Many computerized systems are subject to strict time constraints. Control systems in the oil logging industry, automated manufacture, space exploration, as well as defense, call for fast, time-bound response. If a delay in response beyond the specified time-bound would lead to a system failure (with often dire consequences), the system is classified as a hard real time system.

1.1 Real-Time Software

The needs of these hard real-time systems differ from the common needs addressed by the standard computing environments. Most programming languages abstract functional behavior from timing considerations. Most operating systems and network protocols offer a few time bound services.

This lack of high-level support has lead to many ad hoc approaches. Many time-critical systems have been implemented at assembly level. Higher level implementations have been tested on specific prototypes with common input cases. Others have been subjected to stochastic simulations insensitive to small populations and unstable operating conditions – the essentials of worst case verification. Not surprisingly, such solutions have led to high development costs and unexpected failures.

In contrast, an optimal real-time system should provide a user with programming ease and predictability. The system should accept a high-level specification of real time requirements and verify their feasibility.

Investigated specification approaches vary from integrated program specifications as in real-time languages, to isolated timing specifications. Of the programming languages, the best known is Ada which allows specification of relative constraints. More thorough treatment of real time concepts can be found in research languages such as LUSTRE [12], a synchronous, real time dataflow language. Other approaches range from use of static typing to specify relative and absolute time predicates [13] to the extension of temporal logic to model states and events through clock ticks [14].

Verification efforts vary with the nature of timing constraints. Relative timing constraints enforce sequencing of events within an execution and can be verified without knowledge of machine speeds. Absolute timing constraints place absolute bounds on execution latencies and require knowledge of hardware timing. They are typical of hard real time systems.

Research in verification of relative time constraints has met with much success. Formal specifications such as those based in temporal logic can be used to prove liveness and precedence relations.

Verification of absolute constraints has generated attention at two different levels, at the low machine level and at the high specification level. At machine level, commercial projects have successfully bound system latencies. Masscomp's Real Time Unix [23], for instance, binds system response times through fixed priority scheduling for predictable schedules, through memory locking for processing free of paging and swapping, and through kernel preemption for bound delay of real time processes due to outside system requests.

At higher level, few of the formal specification methods have succeeded in providing a clean interface to the low machine level verification. One of the more successful approaches in this respect has been Jahanian and Mok's real time logic (RTL) [11]. Their logic relies on *safety assertions*, maximum delays along each module, for deadline specification. As long as all assertions are met by the underlying machine, an absolute constraint is feasible. Such assertions hide synchronization and contention costs and correspond to worst case analysis of individual latencies as explored by Leinbaugh and Yamini [5].

More accurate latency bounds can be achieved through direct simulation. However, as Stankovic [1] points out, this approach must tackle the complexity barrier. For all but the simplest programs, accuracy must be sacrificed to lower the cost of computing the simulation.

1.2 Distributed Real-Time Software

The design of a real-time system is further complicated by the frequent use of multiple processors, which may be necessary for several reasons. First, the computing power of a single processor may not be enough to meet hard real time constraints. Second, an application may require different processor types. And third, acquired data may need to be processed at different locations.

As a result, an optimal real time system should provide specification and verification methods within a distributed, heterogeneous environment. This requirement heightens the need for modular timing specifications. It further introduces the need for verifiable real-time communication and its specifications.

1.3 Resource Allocation

A further complication in the design of a hard real time system is the need for an optimized, predictable resource allocation method. A predictable allocation schedule is essential to absolute constraint verification. While easy to achieve, predictability has not been required of many existing schedulers [23].

A reasonably optimized allocation method is essential to meeting absolute constraints. In an optimal real-time system, one would like an automated allocator to arrive at an optimal allocation schedule. Such an allocator would be NP complete even for the much simpler case of two identical processors executing independent tasks with no communication overhead [9]. As a result, all practical scheduling algorithms within a multiple processor environment rely on heuristics. The most common approach is a back-tracking branch and bound search within a simulation. Simplified versions include heuristic transformation of a program graph onto a multiple processor graph, and an independent allocation of computation paths beginning with the most critical path. Several conflicting goals in these approaches are the minimization of complexity, the preservation of a global program view, and the consideration of all relevant time costs.

1.4 Synopsis

In this thesis, we attempt to develop a technique for dealing with real time constraints in the context of a device control and data acquisition system for oil well logging. We narrow our attention to periodic programs and start with an existing software architecture, the Stream Machine [17]. We augment and simplify the computational model to achieve a simple timing specification. We analyze constraints and check feasibility within an allocation scheme.

The content of this thesis tracks the progress of its project. Chapter one introduces the issues and complexity in real time systems and points out related work. Chapter two presents initial thoughts and goals behind this project. Chapter three describes the targeted applications and the inherited programming environment, the Stream Machine. Chapter four presents a specification method for the envisioned real-time costs in our computational model. It outlines our approach towards real-time specification and implements this approach. Chapter five presents a specification method for the envisioned real-time constraints. Chapter six offers a verification method for the developed constraint specifications. It outlines the initial assumptions, and describes and optimizes our verifier. Finally, Chapter seven of the thesis, draws results and lessons from the project and suggests areas of further work.

Chapter 2

Project's Background

2.1 Motivation

The motivation for this work came from the increasing need for feedback control in acquisition of oil well data. On site acquisition and interpretation of oil well data is the main service of the Schlumberger Wireline Testing and Service Companies. Schlumberger acquires and interprets data for clients throughout the world. All acquisition is done with tools and computational resources contained within a highly customized vehicle, the Schlumberger truck. Upon request, the regional Schlumberger branch dispatches a truck to a well site, lowers appropriate sensory tools into the well and acquires data through attached on board computers.

It is essential that well data acquisition be fast and reliable. The acquisition of data halts the production of oil within a well. As a result, the acquisition must be fast in order to minimize the lost revenue and operational expense of an idle well. The malfunction of the sensory tools lowered into the well or of the computational environment can cause delay and loss or damage of expensive tools. As a result, the acquisition must be highly reliable. Finally, the acquired data must be accurate and relevant to further interpretation.

The relevance, accuracy, and reliability of the acquisition process can be enhanced through real-time feedback to the sensory tools. Real-time feedback can increase the accuracy of acquired data as the tool adjusts its speed, resolution, and other parameters based on feedback data. Similarly, real-time feedback can improve relevance of acquired data as the tool zooms in on critical regions of the well and reacts quickly to any aberrations. Finally, real-time feedback can improve reliability through real time monitoring of tool conditions and prompt recovery of an endangered tool.

2.2 The Problem

2.2.1 Feedback

Our process of generating feedback consists of three stages. In stage one, we acquire data from a periodic source. In stage two, we feed the acquired data to an application program and compute feedback data. Finally in stage three, we forward feedback data to its target. Figure 2.1 illustrates the feedback process.

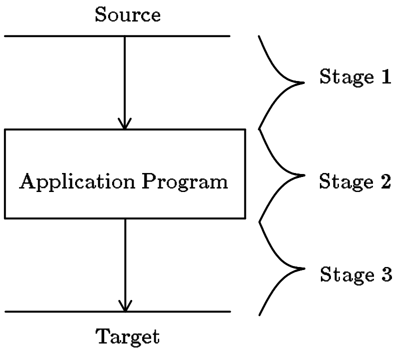

Figure 2.1: The Three Stages of a Feedback Process.

Figure 2.2 illustrates the feedback process within our present domain. In our present domain, the periodic source of data is a sensory tool lowered into an oil well. Data acquired by the tool's downhole processor propagates up the well hole into the Schlumberger truck. On board the truck, the data is accepted by a dedicated acquisition processor. The acquisition processor communicates with an on board workstation via shared memory. Two on board workstations may cooperate in computing feedback data. The target of feedback data is, again, a sensory tool within the well.

We expect our domain to evolve as feedback requirements increase with new sensory tools and as technology progresses. Specifically, we expect to see more computing power on the Schlumberger truck. Multiple and specialized processors and coprocessors will absorb the increased computational load. As the temperature and pressure resistance of VLSI circuits increases, we also expect to see part of the computing stage shifting from the truck into the well. A processor within the sensory tool will reduce the data bandwidth between the tool and the truck and shift low-computation feedback control into the tool.

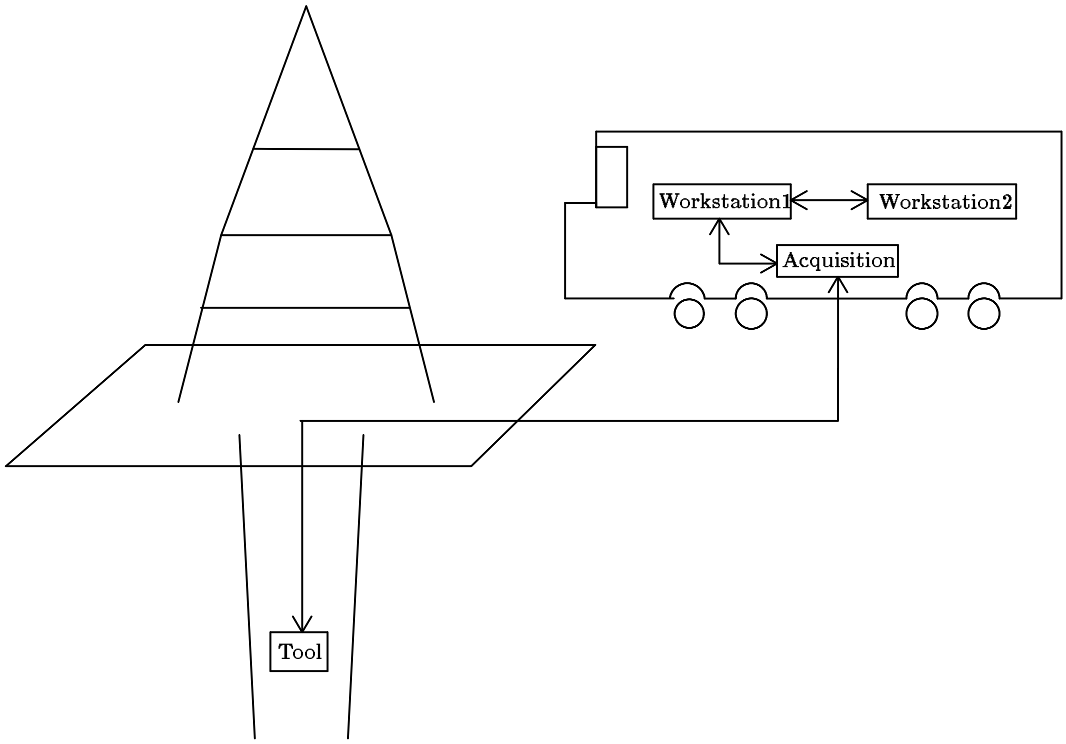

Figure 2.2: Resources Utilized by a Feedback Process within our Domain.

2.2.2 Real-Time

The relevance of feedback information varies with time. For example, a feedback directive to recover a tool becomes irrelevant once the tool has been lost. Similarly, a feedback adjustment of a measurement technique becomes irrelevant once the measurement conditions have changed. To achieve our aims, we must

1. constrain the latency of each feedback process, and

2. guarantee to meet imposed constraints.

To guarantee imposed constraints, we must implement each feedback process and verify that our implementation meets the imposed constraints. To implement a feedback process, we must assign resources to each of the three stages of a feedback process and write the application program of stage 2.

One way to verify a feedback latency constraint is to run and time our feedback process implementation. If the process completes within the constrained time, the constraint has been met. However, this meeting of a constraint does not reflect on future invocations of this implementation. For one, the latency of each invocation may be data dependent. Different input data may require different computation and propagate through the feedback process at different speed. To guarantee a constraint, we would need to run and time our implementation for all possible input values - an unlikely prospect.

Moreover, the individual latencies within an implementation may vary. The latency through a communication channel, for instance, may depend on the instantaneous contention for that channel. The latency of code execution may depend on the momentary number of system call interrupts. While all individual latencies within a feedback process must have a finite upper bound to guarantee a real-time constraint, a single run of the feedback process is unlikely to capture the worst case scenario.

2.3 Goals

The goal of this project is to design a prototype verification system for real-time feedback processes in the Schlumberger oil well logging context. We start with the present feedback process model - the Stream Machine. The goal of our system is to integrate into this model

1. specification of implementation's timing,
2. specification or implementation's real-time constraints, and
3. verification of implementation's constraints.

Chapter 3

Project's Environment

This project builds on top of an existing application domain and an existing feedback process model. The following chapter describes both the application, the Schlumberger well acquisition software, and the model, the Stream Machine. Moreover, the chapter gives examples, extracts their characteristics, and formulates a representative problem used through the remainder of this thesis.

3.1 Stream Machine (SM)

The computational model employed on the Schlumberger trucks is the Stream Machine (SM). The SM imlements a computational model on top of a distributed computer network. An instance of an SM implementation consists of a program description, a machine description, and an allocation description. An SM program consists of buffered communicating sequential processes. A machine consists of distributed hardware resources such as those on the Schlumberger truck. And an allocation maps program components - processes and streams, onto the machine resources - the hardware.

3.1.1 Program

A program consists of a set of processes, or modules, and a set of streams. Modules communicate via tokens along streams. Each module interleaves a finite number of suspending and executing states. Suspended, a module awaits a token along a given input stream. Alternately, a module may await a token along one of several input streams, thus introducing nondeterminism. On the token's arrival, a module consumes the arrived token, and executes. While executing, a module may produce token(s) along any of its output streams. Each stream accepts tokens from exactly one producer module and forwards these tokens to one or more consumer modules. Tokens are guaranteed to reach consumer modules in the order in which they were generated.

Figure 3.1 shows a high-level representation of a module. This module has two input streams, 1 and 3, and two output streams, 2 and 4.

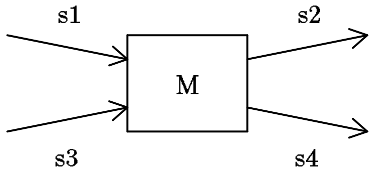

Figure 3.1: A Box and Arrow Representation of an SM Module, M.

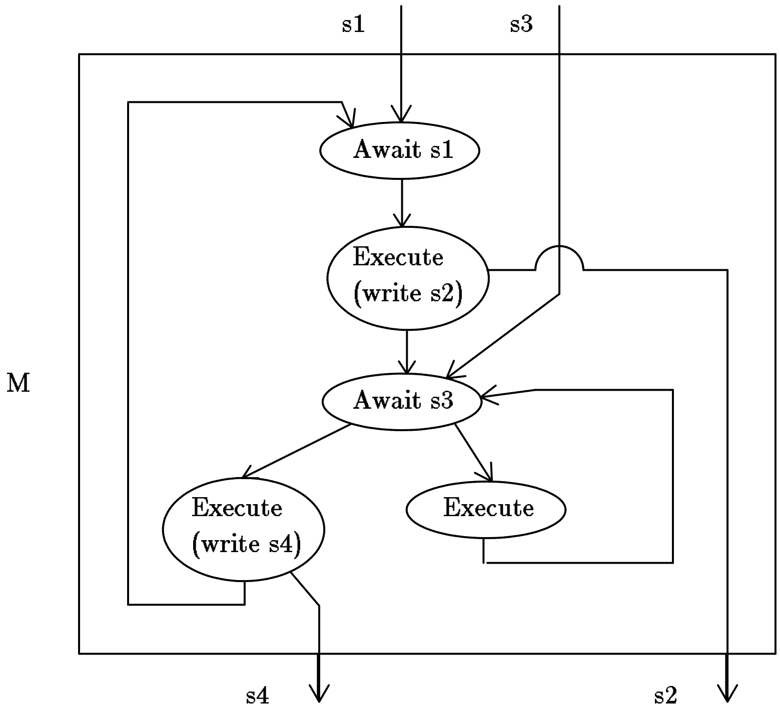

Figure 3.2: A Finite State Representation of an SM Module, M.

Figure 3.2 gives a more detailed view of this module. It shows its internal finite state behavior. In its initial state, this module awaits a token along stream $s1$. It consumes the arrived token along stream $s1$ and executes producing one token along stream $s2$. When done executing, the module awaits a token along stream $s3$. It consumes the arrived token along stream $s3$ and executes. During this execution, the module may produce a token along stream $s4$ and, eventually, return to its initial state. Alternately, the module may produce no tokens and return to its third state, awaiting a token along stream $s3$.

Through the rest of this thesis, we will represent program modules either with finite state diagrams such as that of Figure 3.2, or, more abstractly, with box and arrow diagrams such as that of Figure 3.1.

Assumptions

In order to simplify our specification, we restrict the original Stream Machine model. The Stream Machine model, as described in [17], is a model of buffered communicating sequential processes (CSP). Stream reads and writes are interspersed throughout each module leading to many module states. In each state, a module is either executing with interspersed stream writes or waiting to read from one of its input streams. Figure 3.2 showed an example of possible module states.

We narrow this model by constraining each program module to have only two states - one await state and one execute state. This constraint takes us from a CSP model to a coarse-grain dataflow model. Here each module waits to read from all of its input streams at once. It then executes with interspersed stream writes.

Figure 3.3 reformulates the module of Figure 3.2 into two coarse-grain dataflow modules. This conversion splits the original CSP module along each await state. Note that arrows indicating control flow in Figure 3.2 have now turned into streams. They have become streams 5, 6, and 7. These new streams enforce the original flow of control between what have now become two modules.

We retain a nondeterministic merge module present in the original CSP model as our means of introducing nondeterminism.

Unlike in the original CSP model, in the dataflow model, a module cannot merge tokens from several input streams onto a single output stream in a deterministic order. A standard mechanism for merging tokens in a given order is to specify the desired order along a special input stream, the *Select* stream. The merge module awaits a token along the Select stream and then, based on the token's value, awaits a token along one of its input streams. A dataflow module with a single await state cannot achieve this behavior. It cannot decide which input stream to read next based on the value read along another input stream. We complete our coarse-grain dataflow model by adding a special module which allows this behavior - the deterministic merge module.

The deterministic merge module `determines` the order in which tokens merge onto an

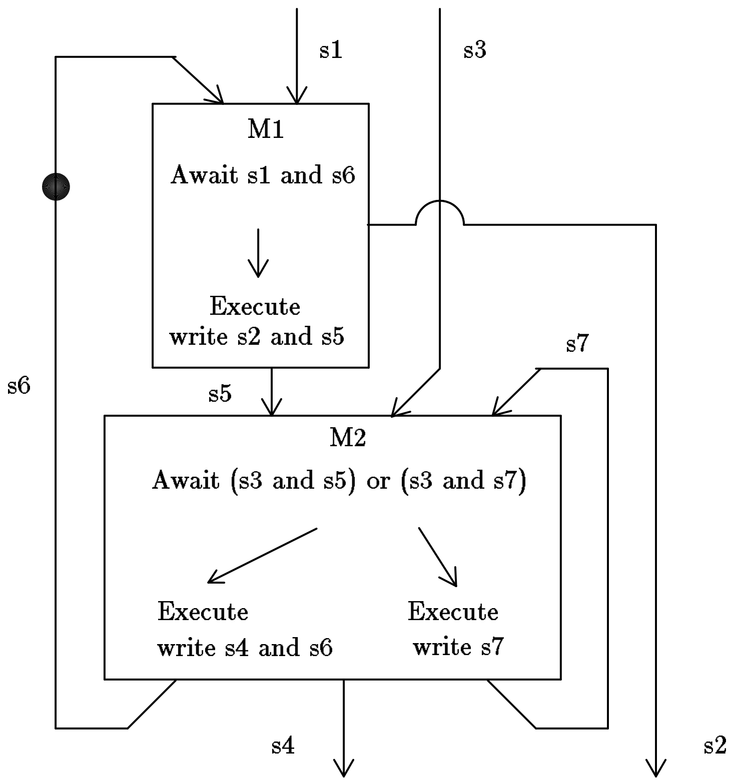

Figure 3.3: A CSP Module, M, Converted into Two Dataflow Modules, M1 and M2.

output stream based on values along the *Select* stream. Compare the simplest merge module with the simplest deterministic merge module (Figure 3.4). Say one token arrives along each input stream sometimes during the program. In case of the merge module, the order in which the two input tokens will merge onto the output stream is not known. It depends on the relative arrival time of the two input tokens. Whichever input token arrives first will merge first. In case of the deterministic merge module the order is known regardless of tokens' arrival time. The order is determined by tokens along a third input stream, the *Select* stream. Tokens along this stream identify the input stream from which to merge next.

3.1.2 Machine

A machine consists of a set of processors and a set of channels. Both, the processors and the channels are heterogeneous. Processor performance is described by the processor's rate of instruction execution, and by the processor's contention protocol. Channel performance is described by the channel's rate of packet propagation, it's latency of propagating a packet, the size(s) of a packet, and the channel's contention protocol. Again, we represent processors and

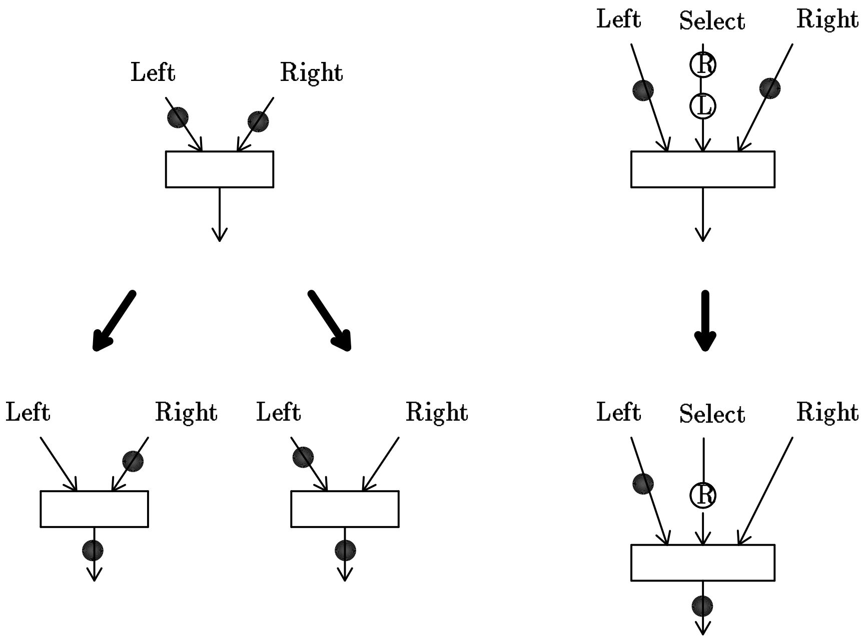

Figure 3.4: Comparison of a Merge Module and a Deterministic Merge Module.

channels with box and arrow diagrams such as that of Figure 3.5. The machine in Figure 3.5 consists of two processors, $P1$ and $P2$, communicating via a bidirectional channel, C. Each box is a processor; each multi- directional arrow is a channel.

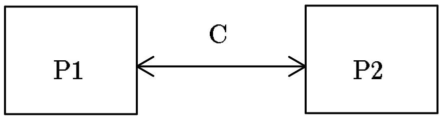

Figure 3.5: A box and arrow representation of a simple SM machine.

3.1.3 Allocation

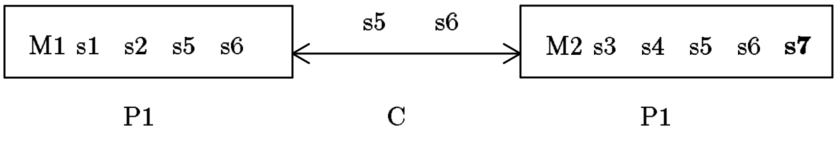

Figure 3.6: A Box and Arrow Representation of an SM Allocation.

An allocation allocates machine resources to program components. We limit our attention to static allocations. Each module is assigned to one processor. Each stream is assigned to a set of processors and channels. We represent allocations with labeled box and arrow diagrams such as that of Figure 3.6. In Figure 3.6, the modules and streams of Figure 3.3 have been allocated onto the machine of Figure 3.5.

Since machine resources are heterogeneous, the performance of each program component depends on its allocation. For modules, the latency of each execution state depends on the processor allocation of that module. Moreover, since specialized processors optimize certain computations, module's execution latencies do not scale with processor's rate of instruction execution. Consider a vector processor, for instance; although its optimal rate of instruction execution may be ten times that of a general processor, a module of scalar code will not execute ten times faster. As a result, the number of high level instructions within a module is insufficient to predict module performance under different allocations.

For streams, the propagation latency of each token along a stream depends on the channel and processor allocations of that stream. Each channel and processor may accept packets of limited length. The latency of a token thus becomes the latency of its packets. Moreover, the time to propagate a token along a channel may vary with each channel. The time to dispatch an arrived or departing packet may vary with each processor. For simplicity's sake, we will assume in all further discussion that each token maps onto exactly one packet. This assumption simply removes a multiplication factor from our discussion.

Aside from individual components' performance, the performance of the entire program also depends on an allocation. It depends on the specific allocations to each resource and on the scheduling method along each resource. Multiple allocations to a resource may cause contention, degrading the program's performance. The resource's scheduling method can moderate this performance degradation by favoring time-critical tasks.

Allocation Constraints

The process of allocating a program onto a machine is limited by three types of constraints: program topology constraints, machine capacity constraints, and dedicated resource constraints. All three types of constraints must be satisfied in order for a program to execute to completion and produced desired results.

Program Topology Constraints These constraints insure that communicating processes will be able to communicate. To achieve this, any two modules which communicate via a stream must be allocated onto a single processor or onto two processors connected by a sequence of channels and intermediate processors.

Machine Capacity Constraints These constraints insure that the limits of each machine component are not exceeded. To achieve this, the load on each processor must not exceed the capacity of that processor. The load on each channel must not exceed the capacity of that channel.

The load on a processor can be determined by scaling all module execution latencies and all packet forwarding latencies along the processor by their frequencies. Similarly, the load on a channel can be determined by scaling all packet propagation latencies along the channel by their frequencies. Finding these frequencies is part of an implementation specification, one of the major goals of our project.

Dedicated Resource Constraints These constraints limit the set of available mappings. They limit the allocation of a given module to certain processors. This limitation is necessary for modules which explicitly make use of certain resources. For instance, a module which displays data on the user's screen must have access to that screen. A module which retrieves data from a sensory tool must have access to that tool.

Real-Time Constraints

The above three constraints guarantee that an allocated program will run to completion and produce desired results. They do not, however, address the real-time behavior of produced results. To address timing properties, we must further constrain an allocation. We place a time limit on the propagation of certain tokens from the creation of token(s) by the source module(s) to the arrival of the **corresponding** feedback token(s) to the target module(s). A data independent specification of this propagation process is the major component of a real-time constraint specification, another major goal of our project.

3.2 Examples

Many programs with real-time constraints are currently in use or under consideration by Schlumberger, with many more anticipated in the future. We give two realistic examples. For future reference, we further develop a sample example encompassing the characteristics of the previous two.

3.2.1 Tool Arm Attachment

The first example is that of anchoring the arm of a tool, the SAT[1] tool, to the wall of a well. This example has been extensively analyzed in [18]. The tool consists of a tester with geophones for measuring seismic vibration and an arm for locking the tool into the borehole. There is a pressure sensor on the arm for detecting when the arm is pressing against the borehole well (Figure 3.7).

The movement from the center of the well towards a wall of the well is controlled by feedback from an application program outside the well. The program responds to two streams of data from the tool:

Distance Stream The distance stream sends up tokens describing the distance of the arm tip from the center of the well.

Pressure Stream The pressure stream sends up tokens describing the pressure on the tip of the arm.

Figure 3.8 shows our implementation diagram for the SAT program. Here, the `Extension` module accepts a token from the `Distance` stream and decides whether the arm has overextended. If so, the module forwards a token to the `Merge` module. Another module, the `Anchorage` module, accepts a token from the `Pressure` stream and decides whether the arm has anchored to a wall. If so, the module forwards a token to the `Merge` module.

A `Merge` module awaits a value along either one of its two input streams. When the `Merge` module receives a token along the `Anchored?` stream, it turns off power to the anchored tool and informs the `Extension` and `Anchorage` modules. When the `Merge` module receives a token along the `OverExtended?` stream, it turns off power to the overextended tool and informs the `Extension` and `Anchorage` modules.

Notice that, in its await state, the `Merge` module awaits a token along any one of multiple streams. As a result, the `Merge` module introduces nondeterminism into our program. It is a nondeterministic merge module.

There are two timing constraints on this program:

- Given a token on the `Pressure` stream, the corresponding token on the `PowerOff`, if any, must arrive back within a time limit sufficient to prevent damage to the arm from pressing against the wall.

[1] Mark of Schlumberger

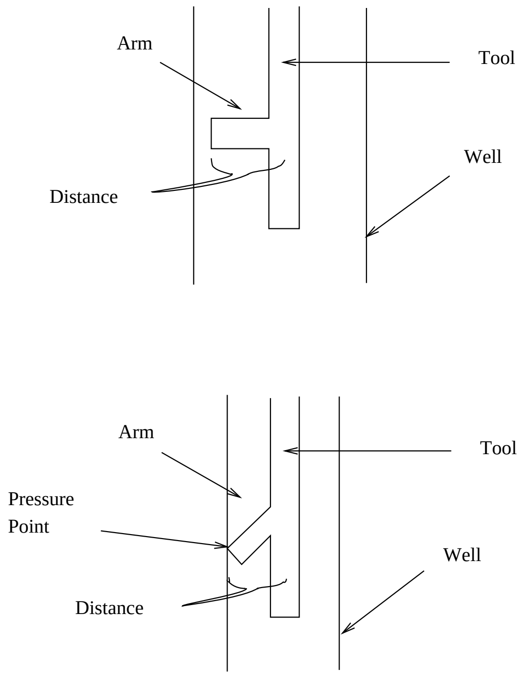

Figure 3.7: An Overextended Arm and an Anchored Arm.

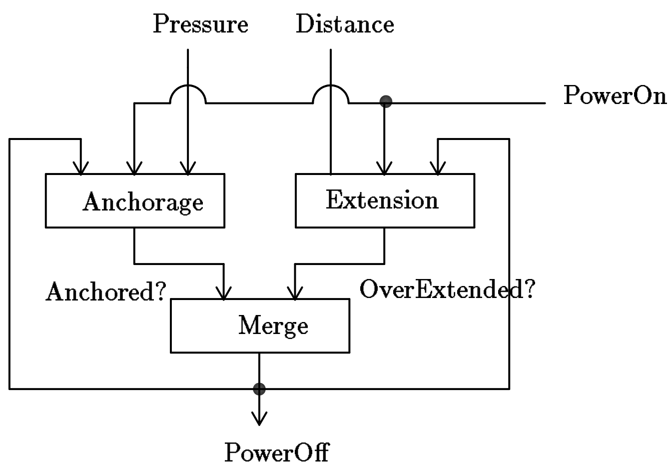

Figure 3.8: SAT Program Diagram.

- Given a token on the `Distance` stream, the corresponding token on the `PowerOff` stream, if any, must arrive back within a time limit sufficient to prevent damage to the arm from overextension.

3.2.2 SLT-L Measurement

A second example of a real-time program is that of acquiring data with the SLT-L[2] tool. This tool, again, consists of a tester lowered into the well. The tester measures the time required for a sound wave to move a certain distance through the rock formation. The measurement is made by using a transmitter to generate a brief sound and a receiver to detect the arrival of sound as it propagates through the formation. The receiver measures the signal's amplitude within a time window that begins after the sound is generated.

Figure 3.9 shows a sound wave in response to sound impulse at `stimulus time`. The sound wave propagates to the receiver `transit time` after its generation by the transmitter. The sound wave is measured within a sliding gate time window. The `sliding gate window` is offset from the stimulus time by a variable time offset.

The quality of the measurement is, again, maintained by feedback from an application program outside the well. The software modules respond to two streams of data from the tool:

Maximum Response Amplitude Stream The amplitude stream sends up tokens describing the maximum amplitude of the response signal.

Signal Transit Time Stream The transit time stream sends up tokens describing the offset of the maximum amplitude response from the stimulus.

Figure 3.10 gives an implementation of the SLT-L program. The `Tool` module in this program is allocated onto the tool processor of Figure 2.2. It provides a periodic source of input data and is the target of feedback data. The `Controller` module is allocated onto the acquisition processor of Figure 2.2. It accepts a packet of data from the tool, separates it into the maximum signal amplitude and the signal transit time, and forwards these to stream 2 and stream 3 respectively. The `Controller` module also accepts feedback data from streams 6 and 7 and forwards these to the tool.

The remaining three modules of Figure 3.10 implement the feedback computation process. The `AmplitudeToGain` module adjusts receiving filter's gain based on the maximum detected

[2]Mark of Schlumberger.

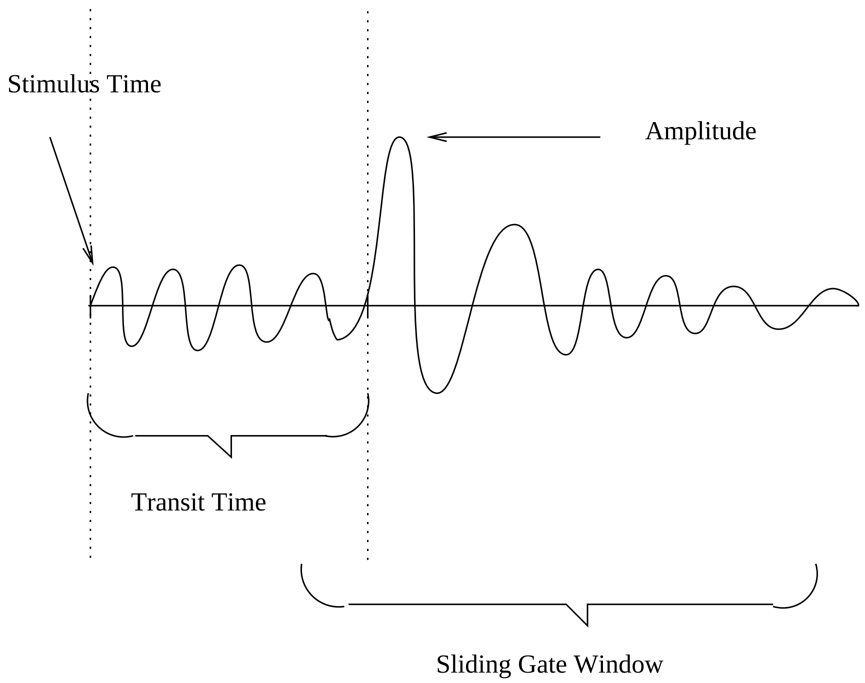

Figure 3.9: SLT-L Sound Wave.

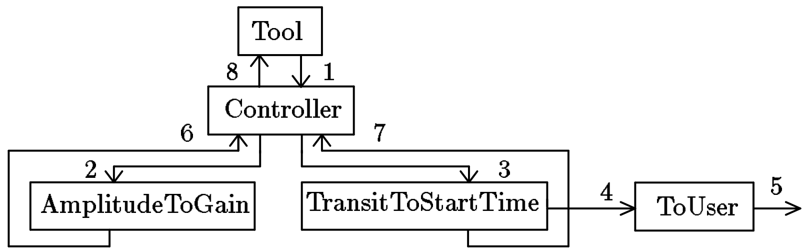

Figure 3.10: SLT-L Program Diagram.

amplitude of the signal. The `TransitToStartTime` module adjust the starting time of the sliding gate window based on the transit time of the previous signal. In addition, the `ToUser` module processes four consecutive transit time measurements and outputs the result to the user.

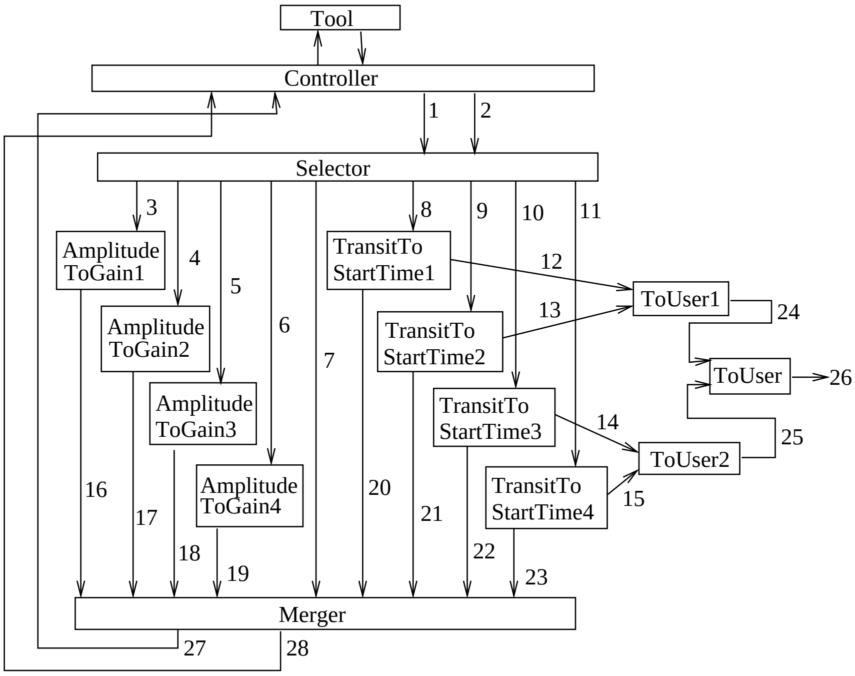

Figure 3.11: Diagram of an Optimized SLT-L Program.

In order to relax the real time constraint on feedback propagation, we present an optimized implementation of the SLT-L program (Figure 3.11). In this implementation, The feedback data is computed in one of four ways corresponding to different transmitter/receiver pairs. The `Selector` module interleaves between the four different ways to compute feedback. In any one cycle, it forwards the maximum signal amplitude to the next `AmplitudeToGain` module. It also forwards the signal transit time to the corresponding `TransitToStartTime` module. In addition, all `TransitToStartTime` modules forward the computed start time to the `ToUser` modules. The `ToUser` modules process four consecutive measurements and forward the result to the user.

This implementation is identical to that of Barstow in [16]. The implementation overlaps the computation of four feedback values. The four-way interleaving of feedback computation lessens the real time feedback constraint. Each amplitude and transit time measurement can be used to adjust the gain and start time of the fourth next measurement instead of the very next one. There are two resulting timing constraints on the optimized SLT-L program:

- Given a token on the maximum response amplitude stream, stream 1, the corresponding token on the filter gain stream, stream 27, must arrive back in time to adjust the gain of the collecting filter for the fourth next measurement.
- Given a token on the transit time stream, stream 2, the corresponding token on the sliding gate start time stream, stream 28, must arrive back in time to adjust the start time for the fourth next measurement.

3.2.3 Sample Program

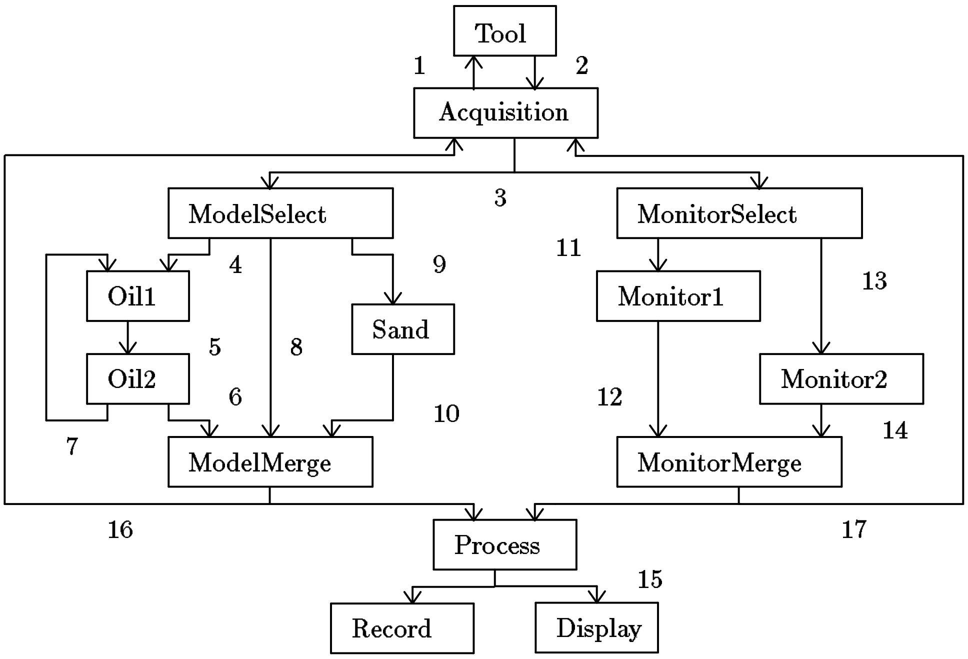

Figure 3.12: SAMPLE Program.

Finally we present an artificial example program, SAMPLE, that is characteristic of our domain and will be used throughout this thesis. Figure 3.12 shows the program diagram of SAMPLE. In this section, we give an informal description of SAMPLE's behavior. A detailed description will follow in Table 4.3.

SAMPLE acquires data with a hypothetical tool. As before, the tool is a tester lowered into a well. SAMPLE monitors the performance of the tool, initiating tool recovery if necessary. At the same time, SAMPLE analyses acquired data. Similarly to the SLT-L program, SAMPLE adjusts tool's measurement parameters based on analyzed data. Moreover, SAMPLE forwards analyzed data for further analysis, storage, and immediate display.

More specifically, SAMPLE provides two types of feedback to the tool through the `Acquisition` module. A periodic feedback signal from the parameter adjusting segment of SAMPLE controls the tool behavior. An emergency feedback signal from the performance monitoring segment of SAMPLE recovers the tool in case of abnormality.

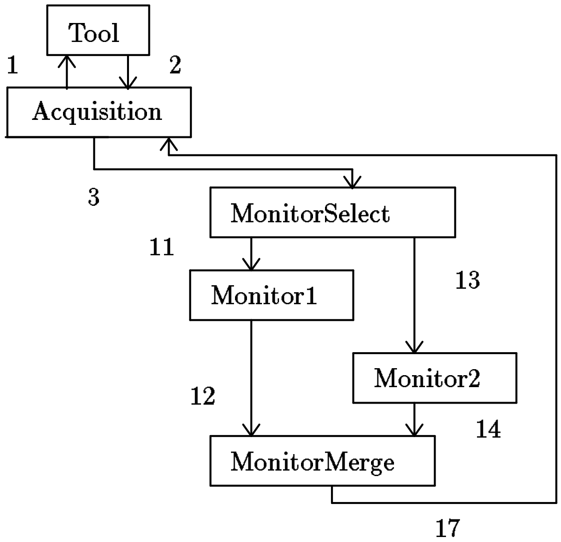

Figure 3.13: The Segment of SAMPLE Program Responsible for Performance Monitoring and Possible Tool Recovery.

Figure 3.13 shows the segment of SAMPLE program responsible for performance monitoring and possible tool recovery. Two modules, `Monitor1` and `Monitor2`, monitor two separate aspects of the tool's performance. Each of these modules evaluates acquired data for certain abnormal conditions and notifies the `MonitorMerge` module of detected abnormalities. Based on input from both modules, the `MonitorMerge` module decides whether to generate an emergency tool recovery signal. Because of `Monitor1`'s long latency, each `Monitor` module only evaluates every other data. The `MonitorSelect` module intermittently forwards data to the `Monitor1` module and to the `Monitor2` module. Correspondingly, the `MonitorMerge` module intermittently merges data from the `Monitor1` module or from the `Monitor2` module.

To illustrate the content of a module, Figure 3.14 shows a possible code routine which comprises the body of the `Monitor1` module. We will return to this routine in the next chapter.

Figure 3.15 shows the segment of SAMPLE program responsible for adjustment of tool's

```
data = read(#11);
if (data <= 42)
   write(#12, ``OK'');
else
   i = 0;
   while (data > 42) and (i < 100)
      data = data + old-data[i];
      i = i + 1;
   write(#12, data/i);
update-old-data(data, old-data);
```

Figure 3.14: Source Code for SAMPLE's `Monitor1` Module.

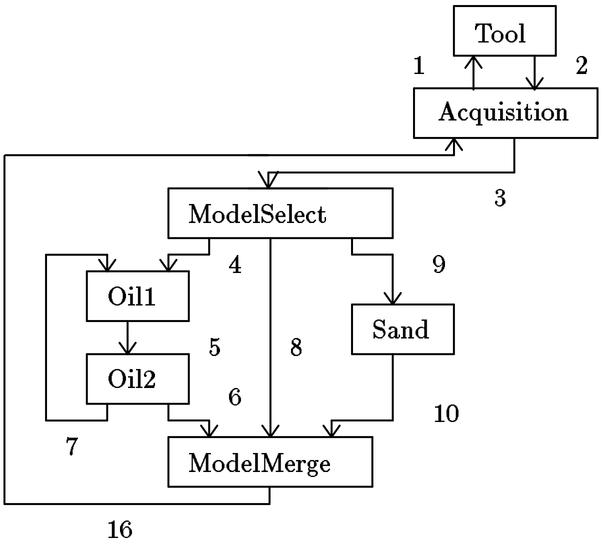

Figure 3.15: The Segment of SAMPLE Program Responsible for Adjustment of Tool's Measurement Parameters.

measurement parameters. The ModelSelect module evaluates incoming data. Depending on data value, the ModelSelect module forwards either one token to the oil model or two tokens to the sand model for further evaluation. Simultaneously, the ModelSelect module informs the ModelMerge module of its model choice along stream 8. Within the oil model, data flows from the Oil1 module to the Oil2 module. The Oil2 module sends evaluated data to the ModelMerge module and update information to the preceding Oil1 module. Within the sand model, two tokens invoke the Sand module. After each invocation, the Sand module forwards a token to the ModelMerge module. The ModelMerge module merges incoming tokens from the two models in the order specified by the ModelSelect module's directives.

Figure 3.15 illustrates a common use of deterministic merge modules. The ModelMerge module - a deterministic merge module, acts together with the ModelSelect module to preserve FIFO (first-in-first-out) ordering of tokens through the subgraph. The two modules preserve the FIFO ordering of multiple tokens entering two different paths, the oil model path and the sand model path. The ModelSelect module informs the ModelMerge module of the order in which it injects tokens into the subgraph. The ModelMerge module merges the outgoing tokens in the order specified by the ModelSelect module.

Stream Machine code for the ModelSelect and the ModelMerge modules (Figure 3.2.3) illustrates this behavior. The code shows that the choice of the second input stream to the ModelMerge module is dependent on the value along the first input stream, the select stream. It illustrates that, while other modules consume a static set of input tokens, a deterministic merge module selects the remainder of its input set based on the **value** of the token along its select stream.

In addition, both the ModelMerge module and the MonitorMerge module forward all output to the Process module for further processing. The Process module sends data to the Display module for immediate display and to the Record module for long term storage.

We constrain both types of feedback in SAMPLE - tool recovery feedback and parameter adjusting feedback. Here, we offer an informal description of these constraints. A formal specification will follow in Figures 5.5 and 5.6.

First, we constrain tool recovery feedback - the time it takes the monitoring segment of SAMPLE to generate a recovery signal. This constraint is conditional on SAMPLE's detection of abnormal conditions. Under normal tool conditions, a recovery signal will, of course, not be generated. Since each of the two performance monitoring modules receives only every other

```
ModelSelect:
data = ...;
if (data == ...)
   write(#8,OilModel);
   write(#4,data);
else
   write(#8,'SandModel);
   write(#9,data);

OilModelMerger:
if (read(#8) == 'OilModel)
   write(#16, read(#6));       %merge from oil model
else
   write(#16, read(#10));       %merge from sand model
```

Figure 3.16: Sample Code for the `ModelSelect` and `ModelMerge` Modules.

data, they will both detect abnormal conditions after two cycles. As a result, we constrain the time it takes to generate a recovery signal to be no more than two cycles. Say, for instance, that the tool's sampling cycle takes 150 time units. Then the time it takes two successive tokens along stream 2 to propagate through the performance monitoring segment of SAMPLE (Figure 3.13) and generate a recovery signal along stream 1 must be less than 300 time units.

Second, we constrain parameter adjusting feedback - the time it takes the parameter adjusting segment of SAMPLE to adjust tool's parameters. This time varies depending on detected formation. An oil rich formation requires different adjustments then a sandy formation. In either case, we constrain SAMPLE to generate adjustment parameters before the tool's next cycle. We assume the tool's sampling cycle to be, again, 150 time units. Then the time before one initial token along stream 2 propagates through the oil or the sand model (Figure 3.15.) and produces one token along stream 1 must be no more than 150 time units.

3.2.4 Program and Constraint Characteristics

We have looked at three different programs in this section: the SAT, the SLT-L, and SAMPLE. From these, we can draw several conclusions about the programs in our domain. First, all three programs contained cyclic paths. These paths were used to provide feedback. Second, all three programs received periodic data from a tool. Finally, the behavior and timing of all three programs depended heavily on input data.

Yet another characteristic of our programs was balanced flow of data. Because of limited buffer sizes, tokens could not accumulate indefinitely along any one arc. Moreover, because of our FIFO model, tokens could not be discarded upon buffer overflow. As a result, the arrival of tokens along the input arcs of a module had to be balanced and consumed steadily. The `MonitorMerge` module of Figure 3.12 illustrated. The module awaited one token along stream 12 for every one token along stream 14.

For each of the three programs we have discussed, we have described real-time constraints which bind the program. Real-time constraints in all three programs also shared several major characteristics. First, constraints were absolute, numeric limits, as opposed to relative, precedence limits. They were often dictated by feedback control rates. In general, constraints specified propagation delay from an initial point to a final point through many possible computation paths.

In addition, the constraints we saw ranged from hard to very soft. A signal to retrieve a malfunctioned multi-million dollar tool was an example of a hard signal. Any chance of missing the constraint limit was a clear failure. In contrast, a high rate signal to adjust tool speed was an example of a soft signal. An unlikely, random chance of missing the constraint limit was acceptable.

Also, constraints could be conditional on branching decisions within the computation. For instance, in our SAT example, the user constrained a critical path of unguaranteed existence, the path from the `Pressure` stream to the `PowerOff` stream. Given a token on the `Pressure` stream, a corresponding token on the `PowerOff` stream is conditional on the `PowerOff` branch of the `Merge` module.

3.2.5 Sample Machine

Having looked at several programs, we next look at a sample machine. Figure 3.17 shows the diagram of a sample machine. Tables 3.2 and 3.1 give the machine's heterogeneous channel and processor parameters. This machine is similar to the wireline acquisition machine of Figure 2.2. It, too, has

- two workstations, `Workstation1` and `Workstation2`, (of uneven capacity)
- an uphole tool processor, the `Acquisition` processor, connected to one of the two workstations, and

- a downhole processor, the `DownHole` processor, connected to the uphole processor.

This sample machine is typical in its lack of homogeneity. It is composed of diverse processors with varying hardware (speed) and system level parameters (multitasking, interprocess communication, scheduling method ...). Channels connecting individual processors are equally diverse. Channel hardware (speed, latency) and system level parameters (broadcast, one way communication, ...) vary.

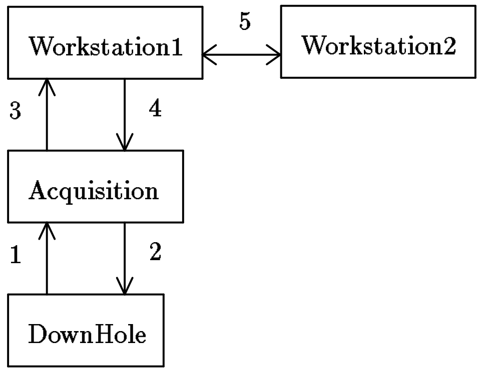

Figure 3.17: Sample Machine.

Processor Parameters		
#	Name	Available Capacity
1	DownHole	100%
2	Acquisition	100%
3	Workstation1	80%
4	Workstation2	100%

Table 3.1: Parameters for Sample Machine Processors.

3.2.6 Sample Allocation

We conclude our examples with a sample allocation of our sample program onto our sample machine. Figure 3.18 illustrates. It shows the allocation of individual program modules to processors. Program's streams have been allocated so as to connect each producer module with all of its consumer modules. As we see from the figure, the number of processes and streams greatly exceeds the number of processors and channels, leading to resource contention.

Any allocation of the sample program onto the sample machine is constrained by three

Channel Parameters				
#	Name	Packet Propagate Latency	Direction From	Direction To
1	SignalUp	4	1–2	
2	SignalDown	4	2–1	
3	SharedMemoryUp	0	2–3	
4	SharedMemoryDown	0	3–2	
5	TruckNetwork	10	5,6–5,6	

Table 3.2: Parameters for Sample Machine Channels.

dedicated resource constraints. First, the `Tool` module must be allocated to the `Downhole` processor. (Conversely, no module other than the `Tool` module may be allocated to the `Downhole` processor.) Second, the `Acquisition` module must be allocated to the `Acquisition` processor. And third, the `Display` module must be allocated onto the `Workstation2` processor.

It is easy to see that our sample allocation satisfies dedicated resource constraints. The `Tool` and the `Acquisition` modules are the sole occupants of their dedicated resources, the `Downhole` and the `Acquisition` processors. And the `Display` module has been correctly allocated to `Workstation2`. The allocation also satisfies topology constraints. All communicating modules are able to communicate via connecting channels and processors. To satisfy machine capacity constraints, we need to determine the maximum load on each processor and channel. This information will easily follow from our verification of real-time constraints in chapter 6. In the remainder of this thesis, we will consider whether this allocation satisfies sample program's real-time constraints.

3.3 Summary

Before attempting a specification of timing costs and constraints, we must gain a practical understanding of our domain. This chapter attempted just that. In doing so, it hinted at several problematic areas.

First, this chapter illustrated the degree to which program's timing depends on input values. In the SAMPLE program, for example, the time to update tool's parameters depended on the formation surrounding the tool. Different calculations were called for in an oily or sandy formation. Even the existence of timing constraints was conditional on input. SAMPLE's

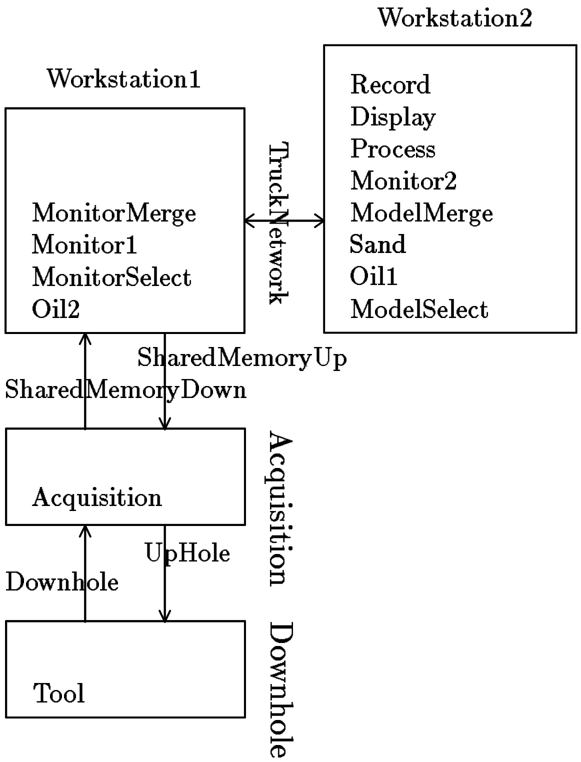

Figure 3.18: Sample Allocation of Program onto Machine.

recovery signal would not be generated without an abnormal status data from the tool.

In addition, allocation affected program's timing. First, the speed of program's modules and streams depended on their assigned processors and channels. Some modules, such as the `Tool` module in SAMPLE, would not run at all under some assignments. Thankfully, we do not attempt to allocate program's resources in this work. However, our timing verifications will have to be allocation dependent.

More seriously, allocation onto limited number of resources indicated timing costs due to contention. In our sample allocation, for instance, eight different modules competed for one processor. With several modules activated concurrently, the contention time could easily exceed the execution time of a module.

Chapter 4

Timing Specification

Having described our domain, we proceed to address the first goal of this project as outlined in Chapter 2 (page 15) - a timing specification of a feedback process. Our goal is to to specify enough timing information in order to verify real-time constraints.

At present, our description of a feedback process consists of program's instructions and its allocation. Take our SAMPLE program. We are given a number of modules and streams (Figure 3.12), together with the source code of each module and with SAMPLE's processor and channel mappings (Figure 3.18).

We are asked to verify whether a constraint is met. Take the simplest constraint through a single module such as SAMPLE's `Monitor1` module. Figure 3.14 showed the source code for this module. Say we constrain the time from the arrival of a token along stream 11 to the creation of one token along stream 12 to be less than x time units. How do we verify this constraint?

Excluding all other costs, the simple time to execute `Monitor1`'s instructions up to and including the generation of a token along stream 12 is not constant. The time varies with the input value read on stream 11. But our imposed constraint must be met for any input value.

Fortunately, we can derive an upper bound. We can derive the maximum possible time to execute instructions up to and including the generation of a token along stream 12. It is the time to execute `Monitor1`'s most time demanding instruction trace on `Workstation1` up to and including a "write(#12,...)" instruction. This is the instruction trace resulting from 100 iterations of `Monitor1`'s `while` loop.

In order to verify the feasibility of generating one token along stream 12 within x time units of an arrived token along stream 11, we have specified the longest execution time separating the two events. The specification amounted to listing the maximum time for the `Monitor1` module to generate a token. In order to verify the feasibility of a constraint through multiple modules, we will need to specify execution times, independent of data values, along all intermediate modules and streams.

We will not tackle timing costs caused by contention in this chapter. Since other timing costs are unaffected by contention, we will postpone discussion of contention until the following chapter. Our aim, by the end of this chapter, will be to specify enough information in order to verify real-time constraints in a contention free program.

We approach our specification design with several goals.

1. **Data Independence:** Most importantly, we wish to avoid data dependent specification. To draw on data values of tokens would be to return to the code-level description of each module and to verification through repeated program execution. Instead, our goal is to statically isolate all possible time events and associated timing costs.

2. **Separation of Program and Machine:** We wish to preserve the Stream Machine's clean separation of program and machine description. Program behavior specification should draw purely on a program; the associated timing specification should draw on an allocated program.

3. **Modularity:** We wish to preserve the modularity of the Stream Machine description. Our program specification should specify behavior and associated timing at component level.

Our first goal is to describe the behavior of a feedback process. We would like to isolate all actions which take time. In case of a stream, the action is clear: it is the propagation of a token along that stream. In case of a module, time consuming actions become less obvious. At each invocation, a module may output tokens along different streams at different times. Its output may depend on its input values as well as on its periodicity. In case of the deterministic merge module, even the invocation time is conditional on which streams the selector stream selects for input. Somehow, we must abstract module's behavior to capture all possible timing costs. Having described all actions which take time, we will then move on to assign timing costs to each action and, finally, to simulate timing and behavior of the entire program.

4.1 Abstracting Behavior of a Module

In each module's invocation, several actions characterize the advancement of time. Take the general module of Figure 4.1. The module is invoked at the moment it accumulates all awaited tokens along streams s_1 though s_m. It consumes its input tokens and executes for some time. At certain times past its invocation, the module outputs tokens along streams s'_1 through s'_n.

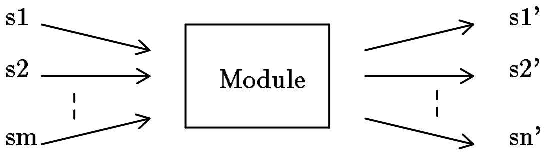

Figure 4.1: Sample Module.

Simple Modules

We start our exploration of behavior with the simplest possible module. This module awaits one token along one input stream and executes outputting one token along one output stream. SAMPLE's `Monitor1` module is an example. `Monitor1` consumes one token along stream 11 and then executes outputting one token along stream 12. Referring to the general module of Figure 4.1, our description simply lists the one input and the one output stream, indicating execution by an arrow ("$\rightarrow$"):

$$s_1 \quad \rightarrow \quad s'_1. \tag{4.1}$$

In case of the `Monitor1` module:

$$s_{11} \quad \rightarrow \quad s_{12}.$$

A simple extension of our description allows for one token along each of multiple input streams and each of multiple output streams. An example of this timing behavior is SAMPLE's `Oil2` module. `Oil2` consumes one token along stream 5 and then executes outputting one token along stream 6 and one token along stream 7. Our extended description simply list all input streams and all output streams, again indicating execution by an arrow ("$\rightarrow$"):

$$s_1 \wedge s_2 \wedge \ldots s_m \quad \rightarrow \quad s'_1 \wedge s'_2 \wedge \ldots s'_n. \tag{4.2}$$

In case of the `Oil2` module:

$$s_5 \quad \rightarrow \quad s_6 \wedge s_7.$$

Finally, a module may await multiple tokens along any one input stream and produce multiple tokens along any one output stream. An example is SAMPLE's `Sand` module which consumes two tokens along stream 9. In our description, we include multiple tokens along a stream by adding an optional coefficient, c, in front of that stream:

$$c_1 s_1 \wedge c_2 s_2 \wedge \ldots c_m s_m \quad \rightarrow \quad c'_1 s'_1 \wedge c'_2 s'_2 \wedge \ldots c'_n s'_n. \tag{4.3}$$

In case of the `Sand` module:

$$2s_9 \quad \rightarrow \quad s_{10}.$$

Selector Modules

The first difficulty arises with data dependent modules. These modules behave differently depending on values of input tokens. They *select* their behavior based on values. Take the `MonitorMerge` module which consumes one token along stream 12 and another one along stream 14. Depending on the values of these tokens, `MonitorMerge` does or does not generate a token along stream 17. As our ultimate goal is a data independent verification, we cannot incorporate token's values into our description. Instead, we describe all possible behaviors, making no choice among them:

$$\begin{aligned} c_1 s_1 \wedge c_2 s_2 \wedge \ldots c_m s_m \quad & \rightarrow \quad c'_{1,1} s'_1 \wedge c'_{1,2} s'_2 \wedge \ldots c'_{1,n} s'_n \\ & \rightarrow \quad c'_{2,1} s'_1 \wedge c'_{2,2} s'_2 \wedge \ldots c'_{2,n} s'_n \\ & \rightarrow \quad \ldots \end{aligned} \tag{4.4}$$

In case of the `MonitorMerge` module:

$$\begin{aligned} s_{12} \wedge s_{14} \quad & \rightarrow \quad s_{17} \\ & \rightarrow \quad . \end{aligned}$$

We will refer to the description of Equation 4.4 as a behavior statement. A behavior statement states how a module will behave for a given input set of tokens.

Merge Modules

Yet another variation on Equation 4.2 captures the behavior of merge modules. In its simplest form, merge modules merge the values along two input streams onto a single output stream. SAMPLE's `Process` module is an example. `Process` awaits one token along stream 16 or one token along stream 17. Whenever a token along either stream arrives, `Process` consumes the token and executes, generating one token along stream 15. The behavior of the `Process` module can be described by two statements:

$$\begin{aligned} s_{16} &\rightarrow s_{15} \\ s_{17} &\rightarrow s_{15}. \end{aligned}$$

In general, the behavior of a merge module can be described by multiple statements which share the same output sets:

$$\begin{aligned} c_{1,1}s_1 \wedge c_{1,2}s_2 \wedge \ldots c_{1,m}s_m &\rightarrow c'_{1,1}s'_1 \wedge c'_{1,2}s'_2 \wedge \ldots c'_{1,n}s'_n \\ &\rightarrow c'_{2,1}s'_1 \wedge c'_{2,2}s'_2 \wedge \ldots c'_{2,n}s'_n \\ &\rightarrow \ldots \\ c_{2,1}s_1 \wedge c_{2,2}s_2 \wedge \ldots c_{2,m}s_m &\rightarrow c'_{1,1}s'_1 \wedge c'_{1,2}s'_2 \wedge \ldots c'_{1,n}s'_n \\ &\rightarrow c'_{2,1}s'_1 \wedge c'_{2,2}s'_2 \wedge \ldots c'_{2,n}s'_n \\ &\rightarrow \ldots \\ \ldots &\rightarrow \ldots \end{aligned} \tag{4.5}$$

Deterministic Merge Modules

We next consider several ways to model the behavior of a deterministic merge module. We have already seen the deterministic merge module's role in preserving FIFO ordering in Section 3.2.3. In the model subgraph in Figure 3.15, the selector module `ModelSelect` together with the deterministic merge module `ModelMerge` maintained the FIFO ordering of tokens entering and exiting the subgraph. To correctly model the behavior of the sample program, we too must preserve this ordering in our specification of modules' behavior.

We start our specification of the deterministic merge module `ModelMerge` from our specification of a simple merge module (Equation 4.5):

$$s_6 \wedge s_8 \rightarrow s_{16}$$

$$s_{10} \wedge s_8 \;\; \rightarrow \;\; s_{16}.$$

Here, an input token on stream 8 matches either an input token on stream 6 or an input token on stream 10, depending on which of the two arrives first. The select/merge pair's FIFO synchronization is simply ignored.

Speculative A slight improvement lets us specify every possible synchronization along our select/merge pair. Since we do not know which set of input tokens a selector token will name, we specify all possibilities. We separate different possibilities with a "|":

$$\begin{array}{cc} & statement_1 \\ | & statement_2 \\ | & \ldots \end{array} \tag{4.6}$$

Within this notation, specification of the `ModelMerge` module becomes:

$$\begin{array}{cc} & s_6 \wedge s_8 \rightarrow s_{16} \\ | & s_{10} \wedge s_8 \rightarrow s_{16} \end{array}$$

The arrival of one token along stream 6 and one token along stream 8 does or does not fire an invocation of the module depending on which of the possible statements we consider.

An obvious disadvantage of this approach is that the behavior of all but one statement is unrealistic. Take the case where the `ModelSelect` module has generated one token each along streams 4 and 8. Choosing `ModelMerge` module's first statement, $s_6 \wedge s_8 \rightarrow s_{16}$, correctly portrays the module's behavior and preserves the select/merge pair's FIFO synchronization. However, choosing `ModelMerge` module's second statement, $s_{10} \wedge s_8 \rightarrow s_{16}$ leads to infeasible behavior. The choice leaves two unconsumed tokens, one along stream 6 and one along stream 8, forever.

Acknowledged We can assert FIFO ordering through an explicit addition of acknowledgment streams to select/merge pairs. Figure 4.2 illustrates on our oil model example. With an added acknowledgment stream, stream 18, the specification of `ModelSelect`'s actions becomes:

$$\begin{array}{rcl} s_3 \wedge s_{18} & \rightarrow & s_4 \wedge s_8 \\ & \rightarrow & 2s_9 \wedge s_8 \end{array}$$

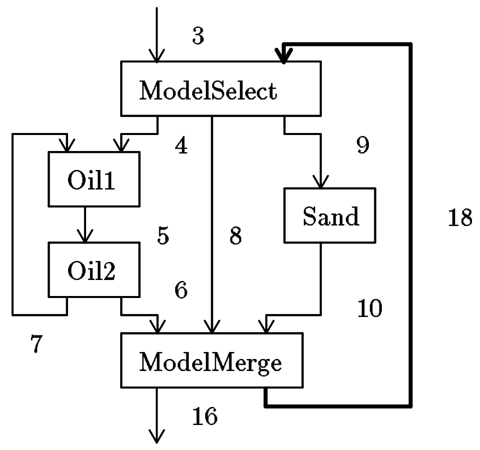

Figure 4.2: The Model Segment of SAMPLE with an Explicit Acknowledgment Stream.

Specification of `ModelMerge`'s actions becomes:

$$s_6 \wedge s_8 \quad \rightarrow \quad s_{16} \wedge s_{18}$$
$$s_{10} \wedge s_8 \quad \rightarrow \quad s_{16} \wedge s_{18}$$

This solution is not optimal. It limits parallelism and lowers execution speed by sequentializing entry into each select/merge pair.

However, in feedback control programs, such as SLT-L (Section 3.2.2), select/merge pairs are commonly used for generality rather than synchronization. In fact, this is the case in our sample program. Here the tool cycle time exceeds the propagation time along either branch of the model segment[1]. As a result, FIFO ordering through this select/merge pair is guaranteed and no acknowledgment is necessary. Because within our application domain explicit FIFO enforcement is often unnecessary, we leave implementation of the alternative tagged approach below for further work.

Tagged A more satisfying approach is the explicit treatment of select/merge pairs. Here we capture the alignment of the select module's and the deterministic merge module's actions. On each invocation of the select module, we tag the generated selector stream token with its output set selection. The corresponding merge module checks the tag of its input token along the selector stream in order to select the remainder of its input set.

We describe the select module as:

$$c_1 s_1 \wedge c_2 s_2 \wedge \ldots c_m s_m \quad \rightarrow \quad s_1' \wedge c_{1,2}' s_2' \wedge \ldots c_{1,n}' s_n', \;\; s_1' = \mathrm{Tag}_1$$

[1]It must in order for feedback to affect the next measurement - a constraint imposed in section 3.2.3

$$\begin{aligned} &\to \quad s'_1 \wedge c'_{2,2} s'_2 \wedge \ldots c'_{2,n} s'_n, \;\; s'_1 = \mathrm{Tag}_2 \\ &\to \quad \ldots \end{aligned} \tag{4.7}$$

Here, tokens along selector stream, s'_1, are assigned a tag designating the select module's choice. Correspondingly, the merge module selects its input set according to the supplied tag:

$$\begin{aligned} \text{If } s_1 == \mathrm{Tag}_1, \quad s_1 \wedge c_2 s_2 \wedge \ldots c_m s_m \;&\to\; c'_{1,1} s'_1 \wedge c'_{1,2} s'_2 \wedge \ldots c'_{1,n} s'_n \\ &\to\; c'_{2,1} s'_1 \wedge c'_{2,2} s'_2 \wedge \ldots c'_{2,n} s'_n \\ &\to\; \ldots \\ \text{If } s_1 == \mathrm{Tag}_2, \quad \ldots \;&\to\; \ldots \end{aligned} \tag{4.8}$$

Like the speculative approach, this solution does not modify the original program. In addition, it is always correct, preserving the intended FIFO synchronization. Table 4.2 shows the tagged behavior specification of our `ModelSelect` and `ModelMerge` pair.

However, the introduction of tags into our description is worrisome. At first glance, it seems that we have violated our main goal - a data independent specification. Tokens along the selector stream clearly carry values from the select module to the merge module. The timing behavior of the merge module depends on the tag value along its selector stream. Are we back to verification through repeated execution for each possible input value? Not quite. Unlike data values, tags do not directly affect evaluation, instead, they align. They align actions of the merge module with those of the select module. Through a finite number of choices, tags describe all valid alignments of those two modules' actions.

Deterministic Merge Modules Summary We have seen two satisfactory ways to express the behavior of deterministic merge modules - through the addition of acknowledgment streams and through tagging. Tables 4.1 and 4.2 show SAMPLE modules' behavior specifications under the two schemes.

State Dependent Modules

In our assumption, we have restricted the Stream Machine modules to two states; each module is either reading its input streams or executing, irrespective of the module's history. This restriction simplified our specification at the cost of lowered performance and expressiveness.

Execution that might have preceded otherwise is postponed until all input streams have been read. And, more importantly, information regarding periodic behavior of a module is lost.

Periodicity played a role in the timing of two of our three illustrative programs. Successive cycles of the SAMPLE program invoked one of two monitoring calculations. In the SLT-L program, successive cycles interleaved among four computations adjusting the tool's parameters.

We illustrate consequences of neglected periodic behavior on SAMPLE's monitor subprogram (Figure 3.13). At this point our best approximation of the `MonitorSelect` module's behavior is:

$$\begin{aligned} s_3 &\rightarrow s_{11} \\ &\rightarrow s_{13}. \end{aligned}$$

This specification states that on each invocation, the `MonitorSelect` module generates either one token along stream 11 or one token along stream 13. The specification does not capture the periodic interleaving of output to streams 11 and 13.

As a result, the subsequent input set specification of module `MonitorMerge`, $s_{12} \wedge s_{14}$, is unrealistic. The specification of the `MonitorSelect` module does not guarantee a balanced arrival of tokens at streams 12 and 14. The `MonitorMerge` module may produce no output and accumulate an overflow of tokens along one of its two input streams. Clearly, this is not the behavior we wished to specify.

To recapture the periodic behavior of modules, we can relax the "statelessness" assumption and allow multiple module states. By convention, we use state 1 as the initial state:

$$\begin{aligned} \text{state x:} \quad & statement_x \\ \text{next state:} \quad & \text{state y.} \end{aligned} \tag{4.9}$$

For instance, the timing actions of the `MonitorSelect` module become:

$$\begin{array}{ll} \text{state 1:} & s_3 \rightarrow s_{11} \\ \text{next state:} & \text{state 2} \\ & \\ \text{state 2:} & s_3 \rightarrow s_{13} \\ \text{next state:} & \text{state 1} \end{array}$$

It is interesting to note that we have not reverted to a CSP model. Our deterministic finite automata of module specifications allow for periodic states. They do not allow for data dependent states. By attaching next state to individual output sets, rather than sets of statements, we could easily reclaim a CSP model. However, none of the sample programs of Chapter 3 indicate a need for this additional source of nondeterminism - a nondeterministic finite automaton.

SAMPLE Modules' Time Critical Actions			
Module	State	Statement	Next State
Tool	-	$s_1 \rightarrow$	-
Acquisition	-	$s_2 \rightarrow s_3$ $s_{16} \rightarrow s_1$ $s_{17} \rightarrow s_1$	-
ModelSelect	-	$s_3 \wedge s_{18} \rightarrow s_4 \wedge s_8$ $\rightarrow 2s_9 \wedge s_8$	-
Oil1	-	$s_4 \rightarrow s_5$ $s_7 \rightarrow$	-
Oil2	-	$s_5 \rightarrow s_6 \wedge s_7$	-
Sand	-	$2s_9 \rightarrow s_{10}$	-
ModelMerge	-	$s_6 \wedge s_8 \rightarrow s_{16} \wedge s_{18}$ $s_{10} \wedge s_8 \rightarrow s_{16} \wedge s_{18}$	-
MonitorSelect	1	$s_3 \rightarrow s_{11}$	2
	2	$s_3 \rightarrow s_{13}$	1
Monitor1	-	$s_{11} \rightarrow s_{12}$	-
Monitor2	-	$s_{13} \rightarrow s_{14}$	-
MonitorMerge	-	$s_{12} \wedge s_{14} \rightarrow s_{17}$ $\rightarrow$	-
Process	-	$s_{16} \rightarrow s_{15}$ $s_{17} \rightarrow s_{15}$	-
Record	-	$s_{15} \rightarrow$	-
Display	-	$s_{15} \rightarrow$	-

Table 4.1: Specification of SAMPLE Modules' Behavior Using Acknowledgment Streams for Deterministic Merge Modules.

Generalization of Merge Modules

Finally, we can expand the specification of merge modules to allow different behavior for different input sets:

$$\begin{aligned} c_{1,1}s_1 \wedge c_{1,2}s_2 \wedge \ldots c_{1,m}s_m &\rightarrow c'_{1,1,1}s'_1 \wedge c'_{1,1,2}s'_2 \wedge \ldots c'_{1,1,n}s'_n \\ &\rightarrow c'_{1,1,2,1}s'_1 \wedge c'_{1,2,2}s'_2 \wedge \ldots c'_{1,2,n}s'_n \end{aligned}$$

SAMPLE Modules' Time Critical Actions			
Module	State	Statement	Next State
Tool	-	$s_1 \rightarrow$	-
Acquisition	-	$s_2 \rightarrow s_3$ $s_{16} \rightarrow s_1$ $s_{17} \rightarrow s_1$	-
ModelSelect	-	$s_3 \rightarrow s_4 \wedge s_8,\ s_8 = Tag_1$ $\rightarrow 2s_9 \wedge s_8,\ s_8 = Tag_2$	-
Oil1	-	$s_4 \rightarrow s_5$ $s_7 \rightarrow$	-
Oil2	-	$s_5 \rightarrow s_6 \wedge s_7$	-
Sand	-	$2s_9 \rightarrow s_{10}$	-
ModelMerge	-	If $s_8 == Tag_1,\ s_8 \wedge s_6 \rightarrow s_{16}$ If $s_8 == Tag_2,\ s_8 \wedge s_{10} \rightarrow s_{16}$	-
MonitorSelect	1	$s_3 \rightarrow s_{11}$	2
	2	$s_3 \rightarrow s_{13}$	1
Monitor1	-	$s_{11} \rightarrow s_{12}$	-
Monitor2	-	$s_{13} \rightarrow s_{14}$	-
MonitorMerge	-	$s_{12} \wedge s_{14} \rightarrow s_{17}$ $\rightarrow$	-
Process	-	$s_{16} \rightarrow s_{15}$ $s_{17} \rightarrow s_{15}$	-
Record	-	$s_{15} \rightarrow$	-
Display	-	$s_{15} \rightarrow$	-

Table 4.2: Specification of SAMPLE Modules' Behavior Using Tagging for Deterministic Merge Modules.

$$
\begin{array}{rl}
\text{state 1:} & \mathit{statements}_1 \\
\text{next state:} & \text{state } \mathrm{i}_1. \\
& \ldots \\
\text{state s:} & \mathit{statements}_s \\
\text{next state:} & \text{state } \mathrm{i}_s.
\end{array}
$$

Where

$$1 \leq i \leq s$$

and where

$$
\begin{array}{rcll}
\mathit{statement} & \equiv & c_1 s_1 \wedge c_2 s_2 \wedge \ldots c_m s_m & \rightarrow \; c'_{1,1} s'_1 \wedge c'_{1,2} s'_2 \wedge \ldots c'_{1,n} s'_n \\
& & & \rightarrow \; c'_{2,1} s'_1 \wedge c'_{2,2} s'_2 \wedge \ldots c'_{2,n} s'_n \\
& & & \rightarrow \; \ldots
\end{array}
$$

Figure 4.3: Our Specification of Module's Time Critical Actions.

$$
\begin{array}{rcl}
& \rightarrow & \ldots \\
c_{2,1} s_1 \wedge c_{2,2} s_2 \wedge \ldots c_{2,m} s_m & \rightarrow & c'_{2,1,1} s'_1 \wedge c'_{2,1,2} s'_2 \wedge \ldots c'_{2,1,n} s'_n \\
& \rightarrow & c'_{2,2,1} s'_1 \wedge c'_{2,2,2} s'_2 \wedge \ldots c'_{2,2,n} s'_n \\
& \rightarrow & \ldots \\
& \vdots &
\end{array}
\tag{4.10}
$$

This generalized specification of merge modules does more then merge streams nondeterministically. A useful analogy is a set of guarded CSP commands. Each command has a different test, its input set; and potentially a different body, its output set. Since they share the same module, commands are sequentially ordered - no two can execute concurrently. Also, since they share the same module, commands can share variables. Moreover, since they share the same set of potential output streams, commands can nondeterministically merge tokens onto the same stream.

Module's Abstract Behavior Summary

In this section, we have specified all actions within a module invocation which take time. Table 4.1 summarizes our specification. We have captured the time of invocation, by listing all awaited input tokens. We have indicated execution time with an arrow "→". We have included points in time when a module generates an output token by listing all output tokens.

We have extended our original specification of one input set, and a single evaluation thread (an arrow and an output set) to include:

1. multiple output sets due to data dependent modules,
2. multiple input sets due to merge modules,
3. explicit acknowledgment streams due to deterministic merge modules,
4. state dependent specifications due to periodic modules, and finally
5. multiple independent statements due to guarded commands.

Table 4.1 illustrates our final specification of SAMPLE modules' abstract behavior (Figure 3.12).

Of these extensions, the most drastic one was that of multiple output sets. In order to achieve data independent specification, we have replaced data dependent computation with a nondeterministic selection of data independent computation. At a module level, the introduced nondeterminicity was sufficient to model all responses to an input set. However, at the inter-module level, some combinations of modules' responses may be unrealistic. Consider two modules whose actions are aligned. Depending on program's input values, the two modules either both act one way or another. A nondeterministic specification of module's actions will not express the alignment of their actions. We leave the problem of inter-module alignment for future work[2].

4.2 Module's Timing Specification

In order to complete our timing specification of a feedback process, we must extend module's behavior with timing specification. We must assign a timing cost to each action:

1. to the time when a module is invoked,
2. to the time during which a module executes, and
3. to the time when a module generates a token.

Three issues complicate description of timing costs:

[2]The reader may have noticed that tags used to align actions of a select module with those of deterministic merge module could be used to explicitly align actions of any two modules.

1. timing costs due to contention,
2. data dependent timing costs, and
3. allocation dependent timing costs.

The first obstacle are costs due to contention. We cannot specify how long it will take for an invoked module to acquire a processor; how many times during its invocation a module will be preempted; or how many arriving tokens will interrupt its execution. The second obstacle are data dependent costs. The duration of module's execution and, relatedly, the time at which it outputs a token, may depend on the values of input tokens. And finally, aside from contention and data dependence, module's execution also depends on its allocation. As we have discussed in Section 3.1.3, the time to execute module's high level instructions varies non-linearly with the assigned processor.

In assigning timing costs, we will postpone consideration of contention and related runtime costs until the next chapter. We will, however, have to somehow abstract away data dependence and take into account program's allocation.

4.2.1 Simple Timing

As before, we start our timing cost assignment with the simplest module. This module awaits one token along one input stream and executes outputting one token along one output stream. SAMPLE's `Monitor2` module allocated to the `Workstation2` processor is an example. `Monitor2` is invoked when one token arrives along stream 13. Ignoring all contention costs, `Monitor2` then executes for exactly 30 time units. This is the time it takes to execute `Monitor2`'s instructions on `Workstation2`. In case of `Monitor2`, the trace of instructions to execute is constant; it is independent of any data values along stream 13 or internal to `Monitor2`. Ignoring contention costs again, `Monitor2` generates one token along stream 14 exactly 10 time units past invocation. It then continues to execute for the remaining 20 time units, updating its internal state. One way to incorporate this information into our specification of `Monitor2`'s actions is as follows:

$$s_{13} \stackrel{30}{\rightarrow} s_{14}[10].$$

Here, we have taken the time of invocation as our point of reference. We have specified `Monitor2`'s execution latency above its execution arrow. And we have indicated the time from `Monitor2`'s invocation to generation of one token along stream 14, next to stream 14.

In general terms, we have expanded specification of the simplest module from:

$$s_1 \quad \rightarrow \quad s_1', \tag{4.11}$$

to:

$$s_1 \quad \stackrel{L}{\rightarrow} \quad s_1'[l], \tag{4.12}$$

where L is the time from invocation to completion, and l is the time from invocation to generation of one token along stream s_1'.

4.2.2 Data Dependent Timing

4.12 does not describe modules with data dependent timing. Take SAMPLE's `Monitor1` module allocated to the `Workstation1` processor. Much like `Monitor2`, `Monitor1` awaits one token along stream 11 and executes outputting one token along stream 12. However, `Monitor1`'s trace of instructions varies from invocation to invocation. The instructions to execute depend on the data value along input stream 12. The time to execute them varies from 50 to 130 time units. Correspondingly, the time to generate one output token along stream 12 varies from 40 to 90 time units.

Since our goal is a data independent timing specification, we cannot specify the relation between data values and timing. Instead, we incorporate `Monitor1`'s data dependent timing into 4.12 by replacing exact latencies with ranges of latencies. In case of `Monitor1`:

$$s_{11} \quad \stackrel{50-130}{\rightarrow} \quad s_{12}[40-90].$$

In general terms:

$$s_1 \quad \stackrel{R}{\rightarrow} \quad s_1'[r], \tag{4.13}$$

where R is the time range from invocation to completion, and r is the time range from invocation to generation of one token along stream s_1'.

4.2.3 General Timing

A minor expansion lets us specify multiple ranges for multiple tokens along each output stream. Take SAMPLE's `ModelSelect` module. `ModelSelect` may generate two tokens along stream 9:

$$\begin{aligned} s_3 \wedge s_{18} \quad &\rightarrow \quad s_4 \wedge s_8 \\ &\rightarrow \quad 2s_9 \wedge s_8 \end{aligned}$$

The first token along stream 9 will be generated 5 to 12 time units past invocation, the following token along stream 9 will be generated 10 to 15 time units past invocation:

$$\begin{aligned} s_3 \wedge s_{18} &\rightarrow s_4[10-15] \wedge s_8[10-15] \\ &\rightarrow 2s_9[5-12, 10-15] \wedge s_8[10-15] \end{aligned}$$

We simply list time ranges from module invocation to token generation for the two successive tokens along stream 9. We guarantee monotonic generation times for successive tokens along a stream. That is, the generation time of the second token along stream 9 is guaranteed to exceed the generation time of the first token.

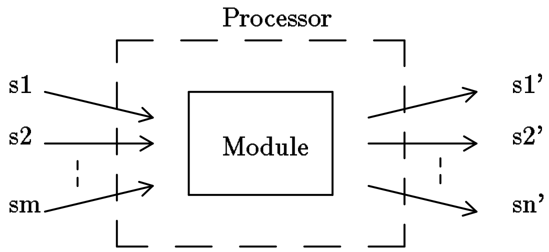

Figure 4.4: Sample Allocated Module.

In general, given a module and its allocation (Figure 4.4), we expand a statement:

$$\begin{aligned} c_1 s_1 \wedge \ldots c_m s_m &\rightarrow c'_{1,1} s'_1 \wedge \ldots c'_{1,n} s'_n \\ &\rightarrow c'_{2,1} s'_1 \wedge \ldots c'_{2,n} s'_n \\ &\rightarrow \ldots \end{aligned}$$

into:

$$\begin{aligned} c_1 s_1 \wedge \ldots c_m s_m &\xrightarrow{R_1} c_{1,1} s'_1[r_{1,1,1}, r_{1,1,2}, \ldots r_{1,1,c_{1,1}}] \wedge \ldots c_{1,n} s'_n[r_{1,n,1}, r_{1,n,2}, \ldots r_{1,n,c_{1,n}}] \\ &\xrightarrow{R_2} c_{2,1} s'_1[r_{2,1,1}, r_{2,1,2}, \ldots r_{2,1,c_{2,1}}] \wedge \ldots c_{2,n} s'_n[r_{2,n,1}, r_{2,n,2}, \ldots r_{2,n,c_{2,n}}] \\ &\vdots \\ &\xrightarrow{R_x} \ldots \end{aligned} \tag{4.14}$$

We indicate the range of latencies from module invocation to completion by associating a range R_i with the ith execution arrow. We expand each $c_{i,j}$ coefficient along an output stream

$$
\begin{aligned}
\text{state 1:} \quad & \mathit{statements}_1 \\
\text{next state:} \quad & \text{state } \mathrm{i}_1. \\
& \ldots \\
\text{state s:} \quad & \mathit{statements}_s \\
\text{next state:} \quad & \text{state } \mathrm{i}_s.
\end{aligned}
$$

Where

$$1 \leq i, j \leq s,$$

and where

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{statement}} \\
\equiv \quad c_1 s_1 \wedge \ldots c_m s_m & \xrightarrow{R_1} & c_{1,1} s'_1[r_{1,1,1}, r_{1,1,2}, \ldots r_{1,1,c_{1,1}}] \wedge \ldots c_{1,n} s'_n[r_{1,n,1}, r_{1,n,2}, \ldots r_{1,n,c_{1,n}}] \\
& \xrightarrow{R_2} & c_{2,1} s'_1[r_{2,1,1}, r_{2,1,2}, \ldots r_{2,1,c_{2,1}}] \wedge \ldots c_{2,n} s'_n[r_{2,n,1}, r_{2,n,2}, \ldots r_{2,n,c_{2,n}}] \\
& \vdots & \\
\ldots & \xrightarrow{R_x} & \ldots
\end{array}
$$

Figure 4.5: Our Final Timing Specification.

s'_j into an array of $c_{i,j}$ latency ranges along stream s'_j. The nth latency range, $r_{i,j,n}$, specifies the range of latencies from module invocation to the generation of the nth token along the output stream s'_j.

4.2.4 Module's Timing Summary

In this section, we have assigned timing costs to module's actions. Ignoring contention overhead, we specified how long it takes to execute module's instructions. The specified time depended on module's allocation. We incorporated data dependent execution by specifying a range rather than a constant time cost. Table 4.3 illustrates our timing specification on our sample program of Figure 3.12.

The reader may have found the derivation of a range, a minimum and a maximum of module's latencies, worrisome. We can obtain a contention free latency, by running a module, alone, on its allocated processor. We can run all branches of each conditional test to obtain the minimum and the maximum latencies. But what about loops? Note that in order to have a finite maximum latency, a module must have a limit on the number of possible iterations. Else, the maximum latency is infinity and no constraints can be met.

SAMPLE Modules' Timing Specification			
Module	State	Statement	Next State
Tool	-	$s_1 \xrightarrow{0}$	-
Acquisition	-	$s_2 \xrightarrow{5} s_3[5]$ $s_{16} \xrightarrow{5} s_1[5]$ $s_{17} \xrightarrow{5} s_1[5]$	-
ModelSelect	-	$s_3 \xrightarrow{15-20} s_4[15] \wedge s_8[15]$ $\xrightarrow{15-20} 2s_9[5-12, 10-15] \wedge s_8[10-15]$	-
Oil1	-	$s_4 \xrightarrow{10} s_5[10]$ $s_7 \xrightarrow{5}$	-
Oil2	-	$s_5 \xrightarrow{5} s_6[5] \wedge s_7[5]$	-
Sand	-	$2s_9 \xrightarrow{10} s_{10}[7]$	-
ModelMerge	-	$s_6 \wedge s_8 \xrightarrow{20} s_{16}[20]$ $s_{10} \wedge s_8 \xrightarrow{20} s_{16}[20]$	-
MonitorSelect	1	$s_3 \xrightarrow{10} s_{11}[10]$	2
	2	$s_3 \xrightarrow{10} s_{13}[10]$	1
Monitor1	-	$s_{11} \xrightarrow{50-130} s_{12}[40-90]$	-
Monitor2	-	$s_{13} \xrightarrow{30} s_{14}[10]$	-
MonitorMerge	-	$s_{12} \wedge s_{14} \xrightarrow{10} s_{17}[10]$ $\xrightarrow{10}$	-
Process	-	$s_{16} \xrightarrow{5} s_{15}[5]$ $s_{17} \xrightarrow{5} s_{15}[5]$	-
Record	-	$s_{15} \xrightarrow{5}$	-
Display	-	$s_{15} \xrightarrow{5}$	-

Table 4.3: Behavior and Timing Specifications for Our Sample Modules.

4.3 Program's Execution Model

Having described the behavior and timing of each module, we are ready to describe the behavior and timing of the entire program. Starting from some initial state, we are ready to simulate the program's abstract behavior through time. For now, our simulation will ignore resource contention costs. In our illustrations, we will assume that each one of SAMPLE's modules has been reassigned to its own processor or each one SAMPLE's streams has been reassigned to its own, fully pipelined channel.

Figure 4.6 illustrates the desired outcome of our simulator. In its initial state, at time 0, the simulator finds one token along stream 2 and one along stream 18. In Figure 4.6, we watch SAMPLE step through time. At each point in time, the simulator displays a snapshot of SAMPLE's program graph, highlights invoked modules, and display tokens along streams. Whenever the simulator encounters several possible behaviors, as at time 19, it forks a separate simulation for each possibility.

In this section, we will describe a simple simulator of abstract program behavior. The purpose of this description is to provide a simple execution model for SM programs. The reader should be aware that no attempt has been made at optimization, only at clarity and simplicity.

The simulator starts with a static description of the program and its allocation. It creates runtime structures to maintain dynamic state of the simulation. Starting from initial state, the simulator then simulates successive events in time, updating its dynamic state with each event.

4.3.1 Static Description

The simulator is initially presented with a static description of the program. This description embodies all the information which we have accumulated about a program and its allocation. For the program, the simulator maintains the program graph: the name of every module and every stream and their interconnection (Section 3.1). In addition, for each module, the simulator maintains its abstract behavior and its timing under the allocation (Section 4.1). For each stream, the simulator maintains token's propagation latency along that stream under the allocation. Since the simulation is free of resource contention, the simulator need not know about the underlying machine.

As an example of a static description, consider again our allocated SAMPLE program.

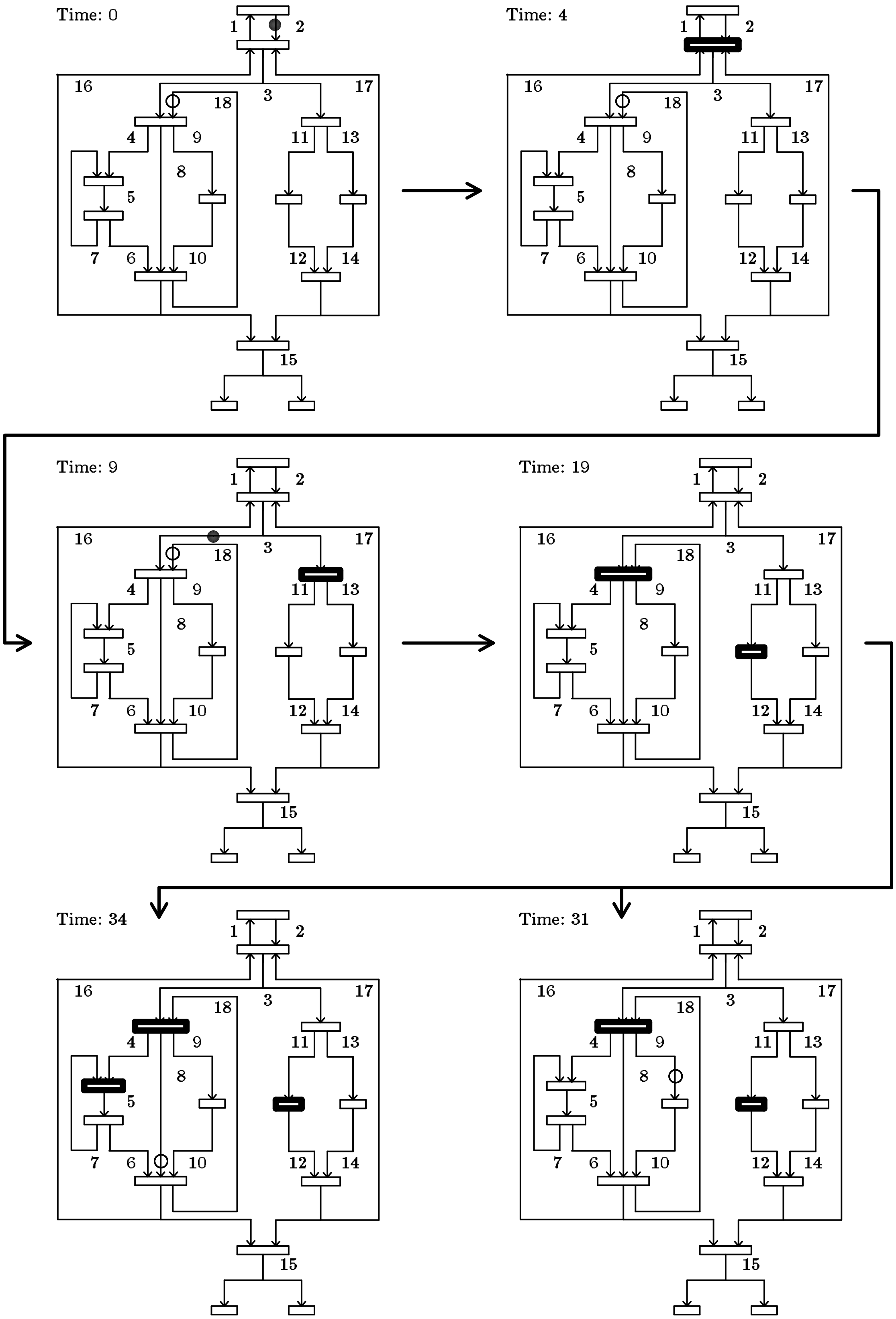

Figure 4.6: Part of Feedback Constraint Verification.

Its static description will consist of the program graph of Figure 3.12, together with streams' latencies under SAMPLE's allocation (Figure 3.18 and Table 3.2), and modules' behavior and timing under SAMPLE's allocation (Table 4.3).

4.3.2 Runtime State

So far, we have outlined the static structures which correspond to our problem description. In addition the verifier must maintain several runtime structures to simulate the passage of time and to accumulate results.

Token

The most obvious runtime state is the token. As the verification progresses, new tokens are created, travel along streams, queue at destination modules, and eventually are consumed. Each of these tokens identifies its stream and its destination module. In addition, each token is stamped with its creation time, the time it was created by the producer module, and later on by its arrival time, the time at which it arrived at the consumer module.

Module

The remainder of runtime state is associated with individual modules. To illustrate, Figure 4.7 shows the runtime state associated with the `ModelMerge` module.

First, each module must maintain input queue(s) for arriving input tokens. As input tokens arrive at modules, they must wait for the remainder of module's input set to arrive. Until then, these tokens must remain queued somewhere. Our simulator associates one token queue with each input stream. In Figure 4.7, the simulator associated one token queue with input streams 6, 8, and 10 of the `ModelMerge` module.

Second, at any point in time, each multi-state module must be aware of its current state. As a new invocation propels the module into its next state, the simulator must update module's state to reflect this change. In Figure 4.7, the `ModelMerge` module is in state 1 - its only state.

Finally, we must insure that previous invocation runs to completion before another one is fired. We insure this by recording the active invocation of each module. Only if there is no active invocation, can a new invocation fire. If another invocation is active, any new invocation must queue on an invocation queue.

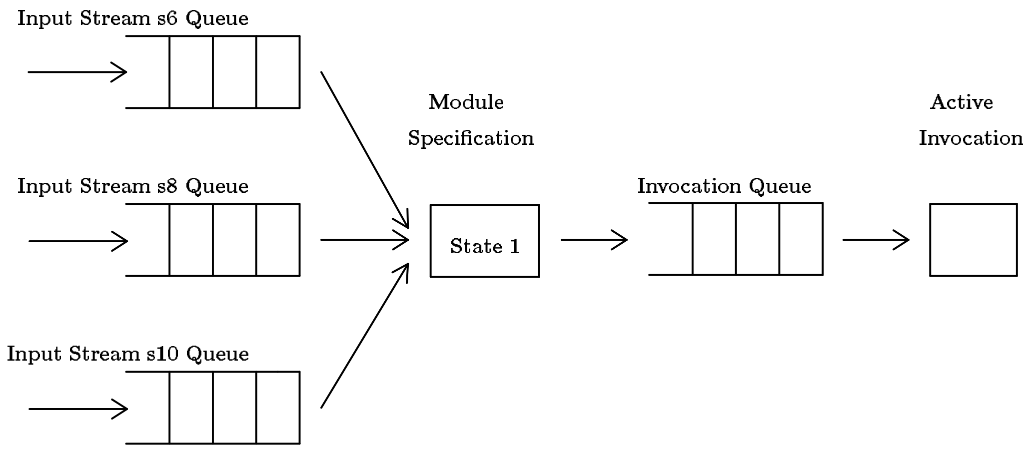

Figure 4.7: A runtime state of the `ModelMerge` module.

4.3.3 Event-Driven Simulation

Having outlined all static and dynamic structures manipulated by our simulator, we are ready to examine the simulation process. The simulation process is event driven. A central event queue maintains a time ordered listing of all pending events.

We will illustrate the simulator's steps on the simulation outlined in Figure 4.6. Figure 4.8 shows successive steps through the event queue as the verification progresses. All successive event queues at one point in time are boxed. Within each box, successive event queues are separated by an arrow. For example, three successive event-queues correspond to time 0. This is because three events occur at time 0.

Each event in Figure 4.8 is described on one line of text. So, for instance, the initial event queue contains two events: the "0 create-token s2" event and the "0 create-token s18" event. The time of each event is written first, followed by the type of the event and its body. For instance, the very first event's time is 0; its type is create-token; and its body is s_2.

Notice that in Figure 4.6, the state of the program at time 0 reflects the state after the last event at time 0 has been handled. This is the case up to time 19. Figure 4.8 stops at time 19, after two separate verifications have been forked.

The overall control flow of the simulation is simple. Starting from some initial state, the simulator loops indefinitely, handling the next event in the time-ordered event queue. At each iteration, it dequeues one event from the event queue and, based on the type of that event, dispatches to the appropriate event handler.

We have already seen one type of an event, the create-token event, above. We now look at

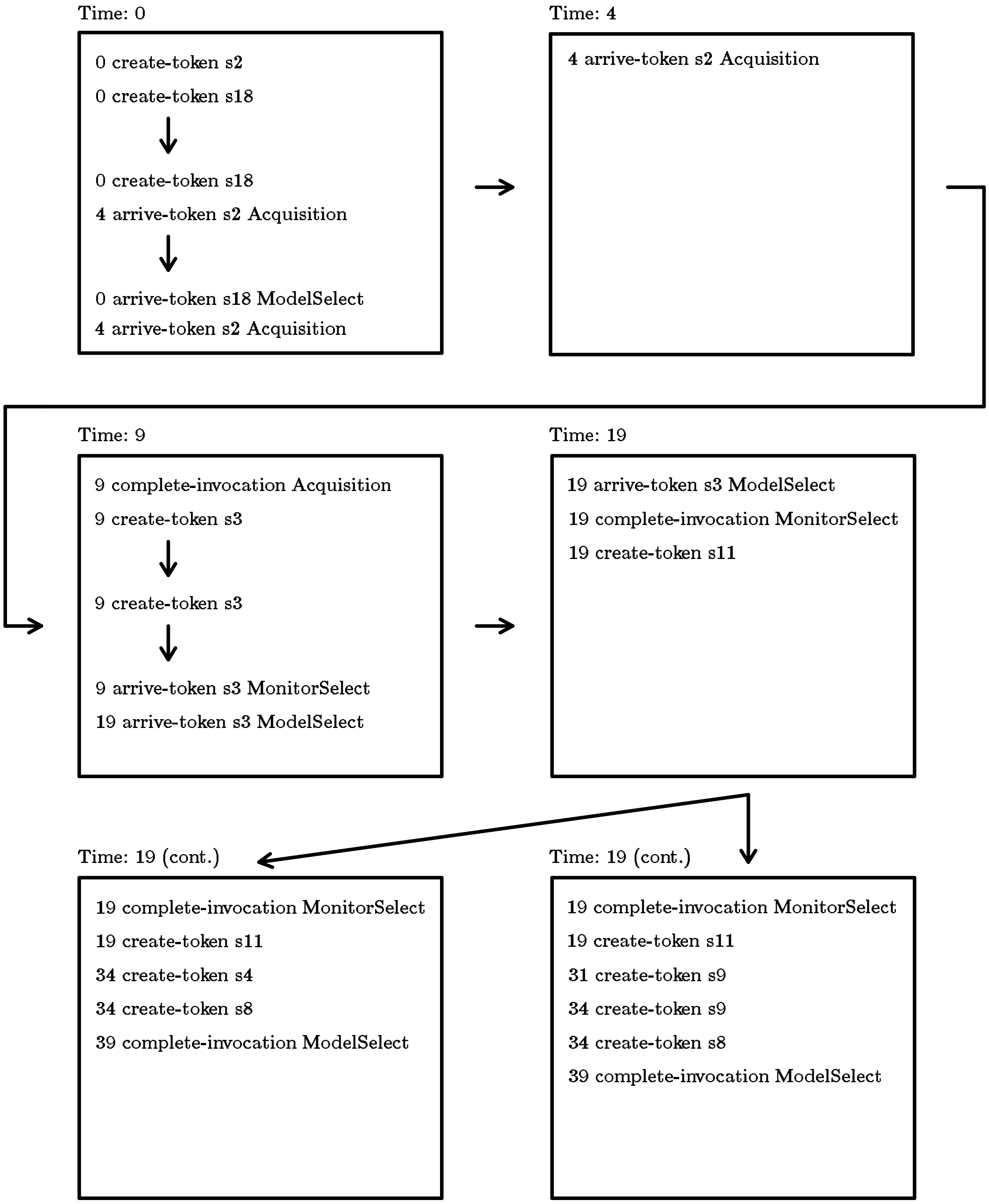

Figure 4.8: Events for Part of Feedback Constraint Verification.

the three different types of events in our simulator and describe the handling of each event type. The two more obvious event types are **creation and arrival of a token**. Anytime a token is created, we must transport the token across a stream to its destination modules. Anytime a token arrives at its destination module, we queue the token and try to fire the module. However, we cannot fire a module invocation before the previous invocation completes. A third event type, **invocation completion**, informs us when a current invocation terminates.

Token Creation

A first type of an event is the create-token event. This event creates a token for each destination module along a stream and transports the newly created tokens to their destination modules. In our simple verifier, we assume that the token need not contend for any resources on its way through the stream. As a result, the time to traverse the stream is constant for each token. We compute each token's arrival time at its destination module and enqueue an arrive-token event at the newly computed arrival time.

Take, for instance, the very first event in the simulation of Figure 4.6: "0 create-token s2". To handle this event, the create-token event handler first finds that the only destination module for a token along stream 2 is the `Acquisition` module and that the time to traverse stream 2 and reach the `Acquisition` module is 4. The event handler creates a token whose creation time is 0, whose destination module is the `Acquisition` module and whose arrival time is 4 (i.e. $0 + 4$). Finally, the event handler enqueues an arrive-token event at time 4. This is the state of the second event queue at time 0.

Token Arrival

A second type of an event is the arrive-token event. Once a token traverses a stream, or arrives, it enqueues onto that stream's input queue. Its arrival triggers an attempt to fire its destination module.

This is the case at time 4 when a token along stream 2 arrives at the `Acquisition` module. First, the token enqueues onto `Acquisition` module's token queue for stream 2. Next, the event handler checks whether `Acquisition` module is ready to fire.

To check whether a module is ready to fire, we compare module's specification in its current state against the contents of input stream queues. We look for a statement whose full input set is queued. In case of generalized merge modules, we may find more than one statement. This is because the arrival of a single token can complete more than one input set. In such case, since our programming model does not specify which statement should fire, we fork a separate simulation for each satisfied statement.

To check whether the `Acquisition` module is ready to fire at time 4, for instance, we look `Acquisition` module's statements in state 1:

$$\begin{aligned} s_2 &\rightarrow s_3 \\ s_{16} &\rightarrow s_1 \\ s_{17} &\rightarrow s_1 \end{aligned}$$

We find that the statement $s_2 \rightarrow s_3$ has a full input set queued. Since only one statement is ready to fire, we need not fork multiple simulations.

Finally, for each statement with complete input set, we dequeue the oldest input tokens and fork one simulation for each possible output set. In case of statement $s_2 \rightarrow s_3$, we dequeue the newly arrived input token from stream 2's queue. Since the statement contains only one output set, $\rightarrow s_3$, we need not fork multiple simulations.

We next check for an active invocation. If an invocation is currently active, we queue the output set on the invocation queue. Otherwise, we fire the output set. In case of the `Acquisition` module, we find that no invocation is currently active, and we fire the output set $\rightarrow s_3$.

To fire an output set, the event handler assigns the output set to the active invocation variable and queues events which immediately result from this invocation. The verifier queues one invocation completion event for the output set and one token creation event for each token within the output set. In queueing these events, the verifier assumes that the invocation will run to completion without interruption. This assumption means that the invocation will not be preempted and that the processing of newly arrived tokens during invocation will not delay the invocation. As a result of this assumption, the time to create tokens and to complete this invocation is constant.

To fire the output set $\rightarrow s_3$ of the `Acquisition` module, the event handler first assigns this output set to be the active invocation of the `Acquisition` module. Next, the event handler queues one complete-invocation event and one create-token event on the event queue. It queues a complete-invocation event at time 9. And it queues one create-token event token for the only output token in the output set, along stream 3, also at time 9.

For another example of a successful attempt to fire, consider the arrival of a token along stream 3 at time 19. The arrive-token event handler queues the token on stream 3's input queue and attempts to fire the `ModelSelect` module. The event handler finds the `ModelSelect` module in state 1 - its only state. It finds one statement whose input set is ready:

$$\begin{aligned} s_3 \quad &\overset{15-20}{\rightarrow} \quad s_4[15] \wedge s_8[15] \\ &\overset{15-20}{\rightarrow} \quad s_9[5-12, 10-15] \wedge s_8[10-15] \end{aligned}$$

The event handler forks two separate simulations - one for each output set. Since the `ModelSelect` module is inactive, both simulations fire their output set. Figure 4.8 shows the state of the event queue in each forked simulation right after the fork.

Invocation Completion

A third type of an event is the complete-invocation event. Once an invocation completes, the module becomes idle and available for new invocations. We empty module's active invocation variable and check whether any invocation is waiting on the invocation queue.

For instance, at time 9, the `Acquisition` module completes its invocation. We reset its active invocation to empty and check its invocation queue. We find that no invocation is waiting to fire and return control to the main loop.

4.3.4 Program's Execution Model Summary

In this section, we have presented an execution model for program's timing based on the abstract behavior and timing specification of individual modules. we have described this execution model in terms of a simple simulator. We have outlined its structure and illustrated its event handling. We will return to this simulator in Chapter 6 where we will extend it in order to verify constraints. At that time, we will outline issues of interest in our design of a realistic simulator.

4.4 Summary

In this chapter, we have introduced a data independent specification of program's behavior and timing. First, we specified all possible behaviors of each module. Next, we assigned timing costs to each module's behavior. At the end, we used the behavior and timing of individual modules to derive a data independent execution model for the entire program.

Chapter 5

Constraint Specification

Having specified program's timing, we focus our attention on our second goal - constraint specification. So far, our notion of a constraint is very informal. We talk of constraints under some circumstances, through some subgraphs. In this chapter, we will try to formalize this notion.

We will refer heavily to the program execution model of Section 4.3. Figures 5.1 and 5.2 show how we can use the execution model to verify a simple constraint. In Figure 5.1, we drop one token onto stream 17 at time 0. In Figure 5.2, we watch SAMPLE step through time. We can end the propagation whenever all awaited tokens arrive at their destination. The awaited tokens are those tokens whose arrival time has been constrained. We refer to them as the constrained tokens. In Figure 5.2, we have ended the propagation when one constrained token along stream 1 arrived at the `Tool` module. Comparing that constrained token's arrival time against some imposed deadline will tell us whether a constraint has been met.

As Figures 5.1 and 5.2 illustrate, we can verify a simple constraint without any formal specification. We can simply visually track all tokens of interest through each step of a simulation. In the next few sections, we will get away from this cumbersome approach.

5.1 Simple Constraints

We start by extracting that information which identifies a simple constraint. Consider again the constraint of Figure 5.1. Here, we have dropped one token onto stream 17 at time 0. Then we stepped through time until one token along stream 1 arrived at the `Tool` module. We stopped our simulation at that point. The information we provided to the simulator could be

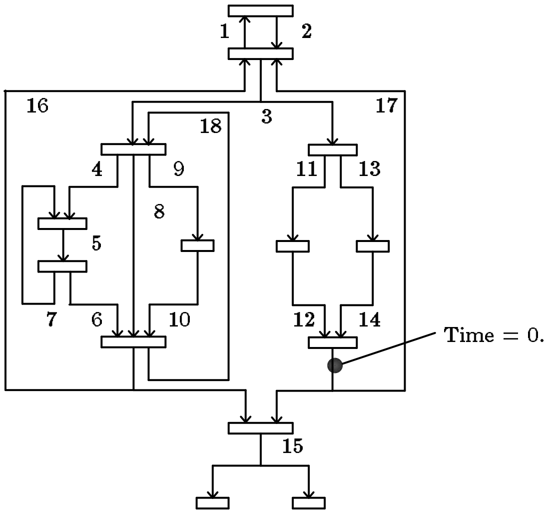

Figure 5.1: Initializing a Simulation.

summarized as:

$$s_{17}[0] \quad \rightarrow \quad s_{1Tool}.$$

Our summary simply lists the stream we dropped a token onto and the time at which we dropped it; together with the input stream and the module at which we awaited arrival of a final token. An arrow, indicating propagation of tokens, separated the dropped initial token from the awaited final token.

We can generalize our summary to any initial token along stream s at time t and to any final token along input stream s' of destination module M:

$$s[t] \quad \rightarrow \quad s'_M. \tag{5.1}$$

To confirm or reject a constraint, we must compare the arrival time of an awaited token against the imposed deadline. Say, in Figure 5.2, we imposed a deadline of 10 time units on the arrival of one token along input stream 1 of the `Tool` module. The simulator's final time, 10, would meet this deadline. We can easily incorporate the imposed deadline into the constraint's specification:

$$s_{17}[0] \quad \rightarrow \quad s_{1Tool}[10].$$

Similarly, we can incorporate a deadline into the general specification of 5.1:

$$s[t] \quad \rightarrow \quad s'_M[d]. \tag{5.2}$$

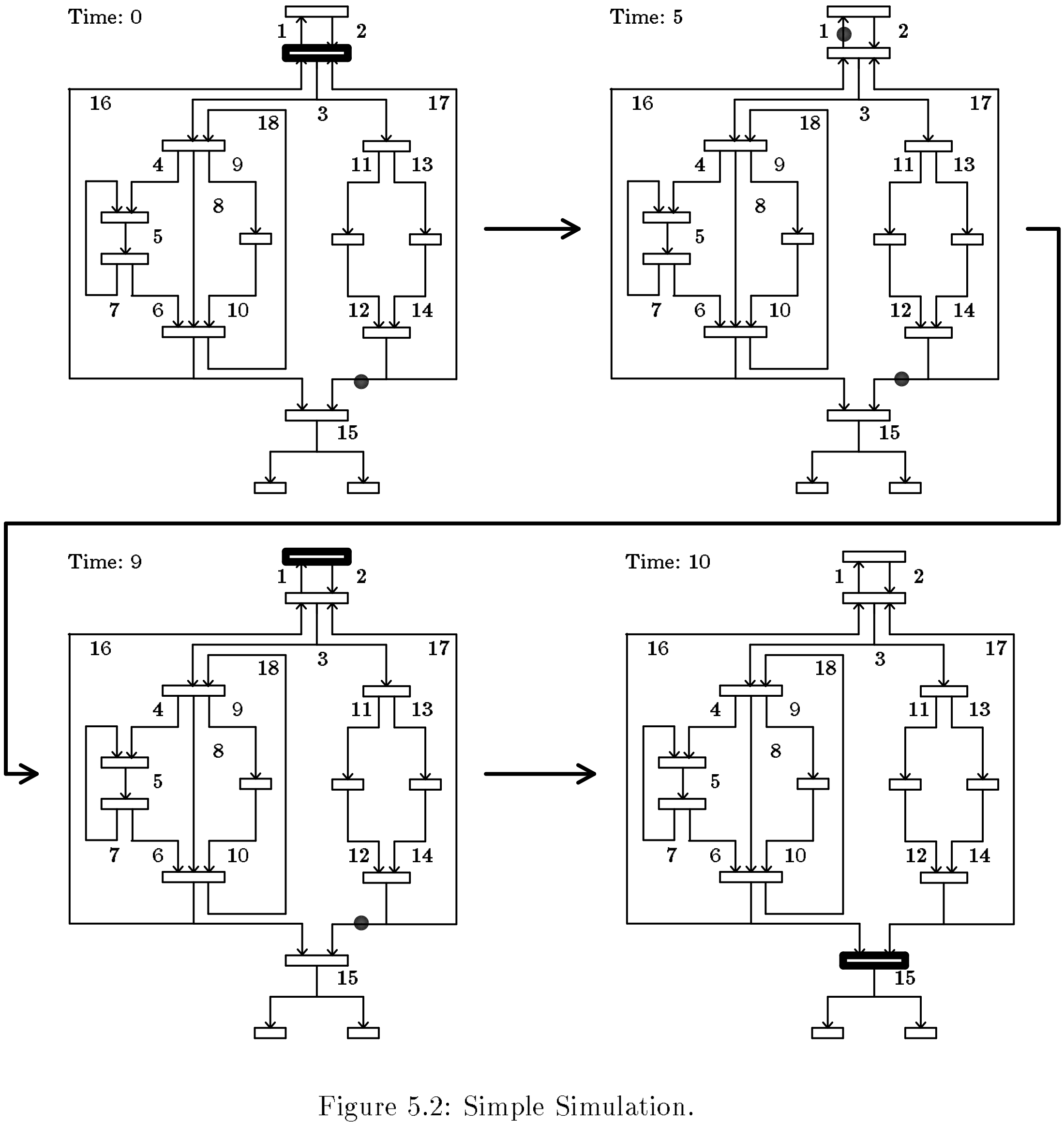

Figure 5.2: Simple Simulation.

Here, we constrain the arrival time of a token along input stream s' of module M to be less than some deadline d.

A simple extension of 5.2 lets us drop multiple tokens along a stream at different times. An example of such a constraint is the tool recovery constraint. A recovery signal can only be generated after two tool cycles. So to observe a recovery token along stream 1, we must drop two successive tokens onto stream 2 one cycle time, or 150 time units, apart:

$$s_2[0, 150] \quad \rightarrow \quad s_{1Tool}[300].$$

In general, we can drop n tokens onto a stream at times t_1 through t_n:

$$s[t_1 \ldots t_n] \quad \rightarrow \quad s'_M[d]. \tag{5.3}$$

By convention, we drop the first token at time 0.

An analogous extension lets us constrain arrival times of multiple tokens at an input stream s' to module M:

$$s[t_1 \ldots t_n] \quad \rightarrow \quad s'_M[d_1 \ldots d_m]. \tag{5.4}$$

Here, we constrain m successive tokens to arrive within m monotonically increasing deadlines d_1 through d_m.

Finally, the reader may have noticed that both of SAMPLE's constraints, the feedback constraint and the tool recovery constraint, will need to drop initial tokens along multiple streams. This is so because stream 18, the acknowledgment stream, must have a token in order to enable SAMPLE's `ModelSelect/ModelMerge` pair (Section 4.1). We can extend our tool recovery constraint to drop an additional token along stream 18 as follows:

$$s_2[0, 150] \wedge s_{18}[0] \quad \rightarrow \quad s_{1Tool}[300].$$

In general, we can extend 5.4 to drop initial tokens along multiple streams:

$$\begin{aligned} & s_1[t_{1,1} \ldots t_{1,n_1}] \wedge s_2[t_{2,1} \ldots t_{2,n_2}] \wedge \ldots \\ & \quad \rightarrow \quad s'_M[d_1 \ldots d_m]. \end{aligned} \tag{5.5}$$

Analogously, we can constrain arrival times of tokens at multiple destinations:

$$\begin{aligned} & s_1[t_{1,1} \ldots t_{1,n_1}] \wedge s_2[t_{2,1} \ldots t_{2,n_2}] \wedge \ldots \\ & \quad \rightarrow \quad s'_{1M_1}[d_{1,1} \ldots d_{1,m_1}] \wedge s'_{2M_2}[d_{2,1} \ldots d_{2,m_2}] \wedge \ldots . \end{aligned} \tag{5.6}$$

5.2 Conditional Constraints

Difficulty arises as constrained paths encounter selector modules. Selector module, such as the `MonitorMerge` module, fire one of several output sets depending on data values. In case of the `MonitorMerge` module, input set $s_{12} \wedge s_{14}$ enables one of two output sets, $\rightarrow s_{17}$ or $\rightarrow$. Having filled `MonitorMerge`'s input set, the simulator forks a separate simulation for each output set.

This may not be the correct behavior with respect to a given program constraint. For each invocation of the `MonitorMerge` module, a given constraint may constrain propagation of tokens through one or through both output sets. A constraint which constrains propagation through both output sets is insensitive to the data dependent actions of the `MonitorMerge` module. Such constraint must be met no matter which output set the `MonitorMerge` module selects.

An example of such a constraint is SAMPLE's feedback constraint. Using 5.6, we specify SAMPLE's feedback constraint as:

$$s_2[0] \wedge s_{18}[0] \quad \rightarrow \quad s_{1Tool}[150].$$

Consider the trace of feedback constraint's verification. We drop one token onto streams 2 and 18 at time 0. In Figure 4.6, we watched them propagate. When a token arrived at input stream 3 of the `ModelSelect` module, it enables one of two threads: $\rightarrow s_4 \wedge s_8$ or $\rightarrow 2s_9 \wedge s_8$. Since we wished to constrain a feedback signal through both the oil and the sand model, we forked two simulations.

In contrast, a constraint which constrains propagation through only one of module's several output sets is conditional on the data dependent actions of that module. We call such a constraint a **conditional** constraint. SAMPLE's tool recovery constraint (Section 3.2.3) is a conditional constraint. It is conditional on the `MonitorMerge` module's selecting thread $\rightarrow s_{17}$. To verify such a constraint, we need to inform our verifier which output set to pursue.

As Figure 4.6 illustrated, we can straightforwardly extend our simulator to verify conditional constraints. We simply have the user inform the simulator which threads to pursue further. But how do we specify conditional constraints independent of simulation?

First, we can declare the selection of all output sets to be the default. Whenever we do not explicitly state otherwise, the simulator will pursue all available evaluation threads. Under this default assumption, 5.6 suffices to specify SAMPLE's feedback constraint:

$$s_2[0] \wedge s_{18}[0] \quad \rightarrow \quad s_{1Tool}[150].$$

The selection of all threads along the `ModelSelect` module has become implicit.

But we still need to express conditional constraints such as the tool recovery constraint. Specifically, we need to express the fact that the tool recovery constraint applies only in case the `MonitorMerge` module selects to output a token along stream 17. One approach is to incorporate `MonitorMerge` module's selection into the simulation. We simply redefine the actions the `MonitorMerge` module for the purposes of the tool recovery constraint. We narrow the choice of evaluation threads within its first invocation to $\rightarrow s_{17}$:

$$\begin{array}{lccc} \text{state 1:} & s_{12} \wedge s_{14} & \rightarrow & s_{17} \\ \text{next state:} & \text{state 2} & & \\ & & & \\ \text{state 2:} & s_{12} \wedge s_{14} & \rightarrow & s_{17} \\ & & \rightarrow & \\ \text{next state:} & \text{state 2.} & & \end{array}$$

As we wished, the simulator will now verify the tool recovery constraint only for the case where the `MonitorMerge` module selects to output a token along stream 17.

Our corrected specification of the tool recovery constraint becomes:

$$s_2[0,150] \wedge s_{18}[0] \quad \rightarrow \quad s_{1Tool}[300].$$

and

$$\begin{array}{llccc} MonitorMerge & \text{state 1:} & s_{12} \wedge s_{14} & \rightarrow & s_{17} \\ Module & \text{next state:} & \text{state 2} & & \\ Timing & & & & \\ Specification & \text{state 2:} & s_{12} \wedge s_{14} & \rightarrow & s_{17} \\ & & & \rightarrow & \\ & \text{next state:} & \text{state 2.} & & \end{array}$$

To extrapolate, we can specify any conditional constraint by redefining the actions of some selector modules. For successive invocations of these modules, we narrow the choice of output sets, using states.

5.3 Nondeterministically Merging Constraints

The simulator of Figure 4.6 still cannot verify some constraints. It cannot verify those constraints which propagate through nondeterministic merge modules. When a constrained token propagates through a nondeterministic merge module, it may merge onto the same stream as other unconstrained tokens and lose its identity.

Unfortunately, both of SAMPLE's constraints, the tool recovery constraint and the feedback constraint, propagate through a nondeterministic merge module - the `Acquisition` module. Take the tool recovery constraint. To verify this constraint, we drop two tokens onto stream 2 at times 0 and 150; and one token onto stream 18 at time 0. We then watch these tokens propagate through our timing specifications. Figure 5.3 shows four snapshots of our simulation. We want to end this propagation when a recovery token has propagated through SAMPLE's monitor segment (Figure 3.13) and reached the `Tool` module along stream 1. Unfortunately, two feedback tokens, having propagated through SAMPLE's model segment (Figure 3.15), will also reach `Tool` module along stream 1.

As Figure 5.3 illustrates, our simulator gives us no indication which of the three tokens has arrived. Since we cannot tell which of the three tokens has reached the `Tool` module, we cannot tell whether to end our simulation. If the arrived token is the awaited recovery token, then we would like to end our simulation and compare x against the imposed deadline. On the other hand, if the arrived token is one of the two feedback tokens, then we would like to continue our simulation, awaiting a recovery token. We must somehow identify the arrived token.

Our dilemma was brought about by the `Acquisition` module. This module has nondeterministically merged two feedback tokens from SAMPLE's model segment and one recovery token from SAMPLE's monitor segment onto stream 1. Once on stream 1, these tokens became indistinguishable. To recover the identity of these tokens, we can **tag** them, much like airport luggage, as they enter the `Acquisition` module.

Figure 5.4 demonstrates our modified simulation. Here, when a token reaches the `Acquisition` module, the module updates the incoming empty tag to indicate token's origin. It specifies that the token along its output stream, stream 1, was initiated by one token along its input stream 16 or input stream 17. Consequently the tokens arrive, tagged and identified, at the `Tool` module.

In general, we can extend our simulator to update token's tag each time it invokes a nondeterministic merge module, merging that token onto a shared stream[1]. By looking at the tag, the user will then be able to identify each token. In case of the tool recovery constraint the

[1]Note that not all invocations of the nondeterministic merge module `Acquisition` merge tokens onto a shared stream.

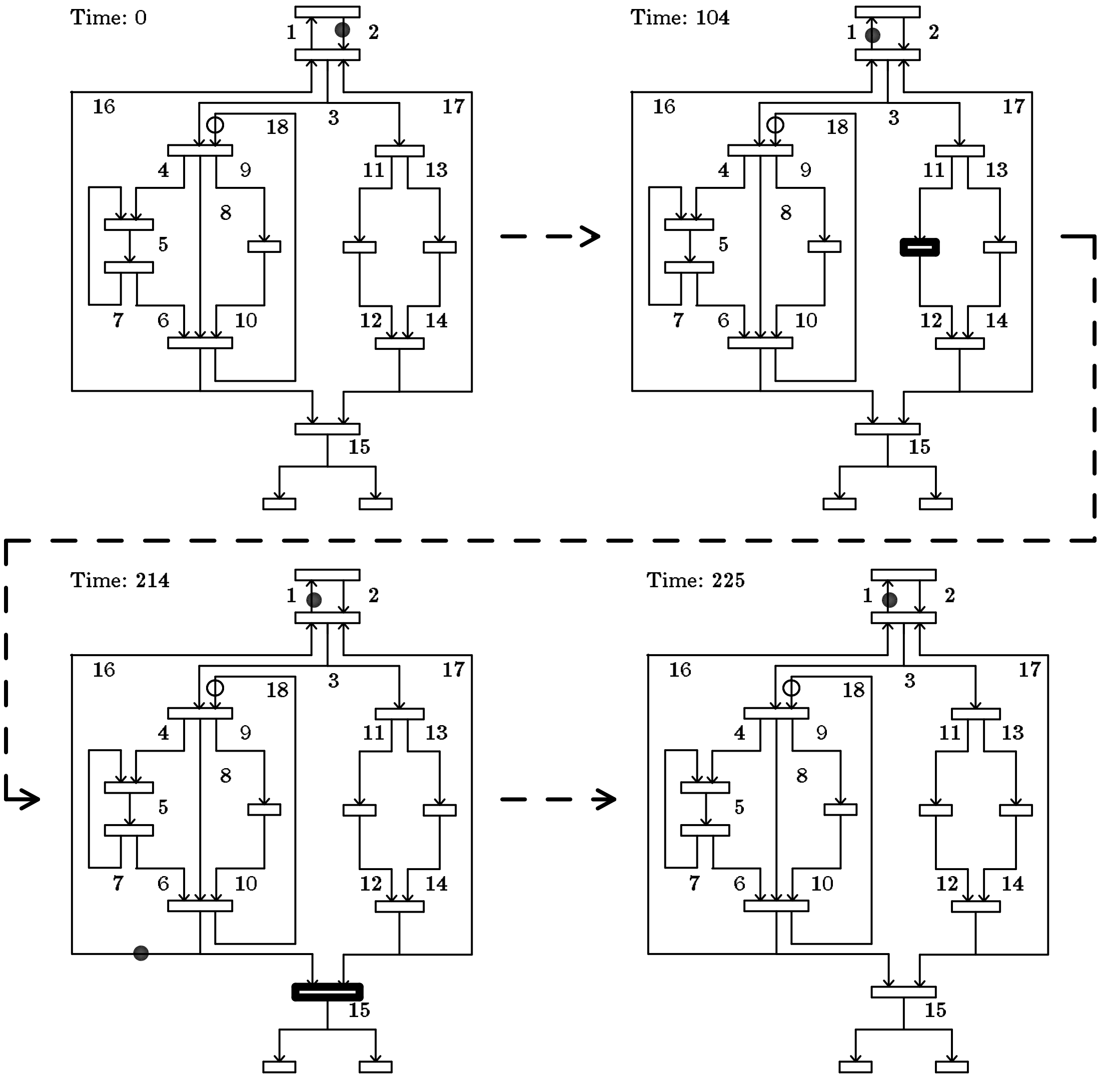

Figure 5.3: Tool Recovery Constraint Simulation.

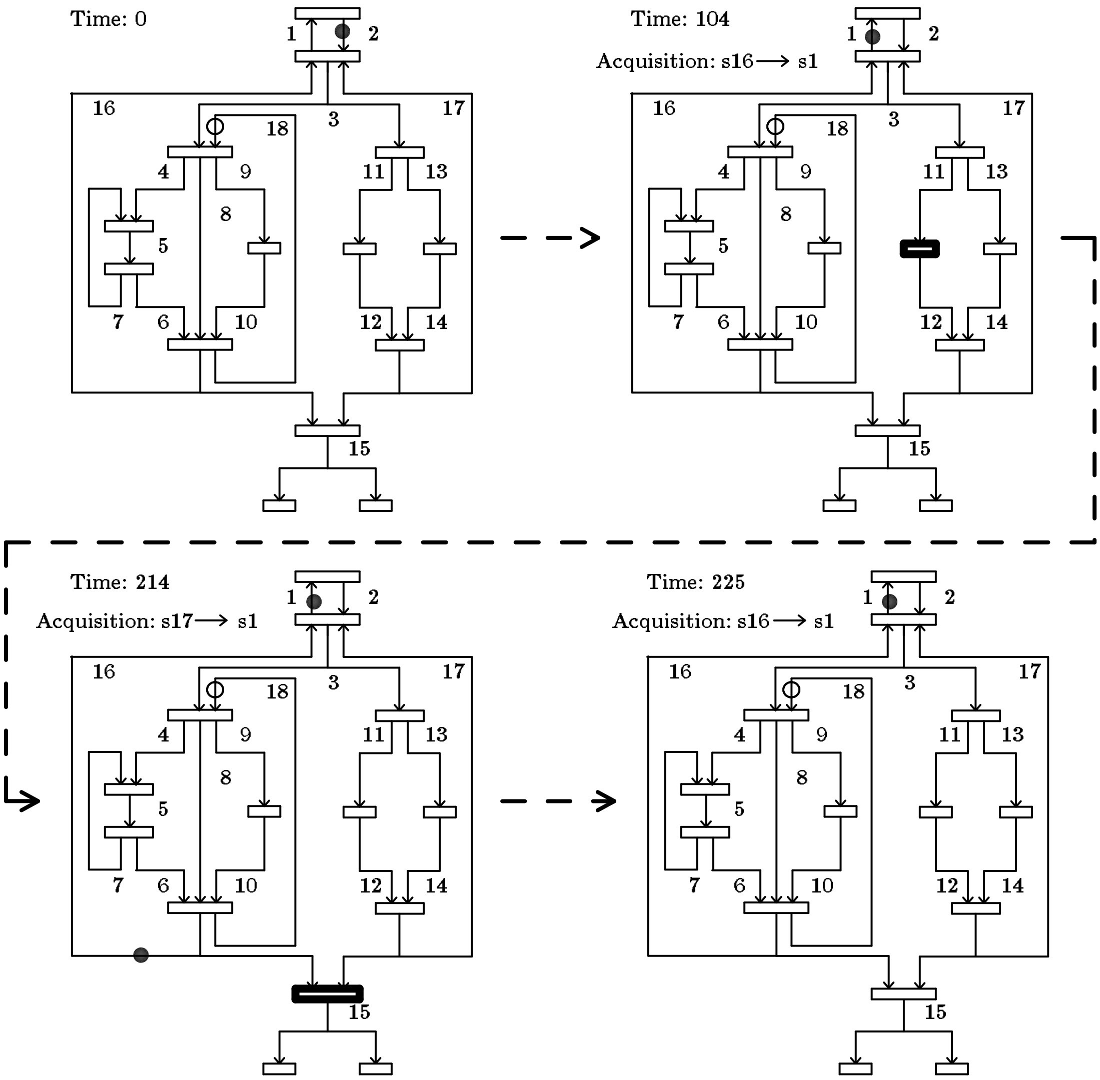

Figure 5.4: Modified Tool Recovery Constraint Simulation: Displayed Next to Each Token is the Token's Tag.

user will know whether a recovery token or a feedback token has reached the `Tool` module.

But how do we specify constraints through nondeterministic merge modules independent of simulation? As we have just seen, in the presence of nondeterministic merge modules, the simulator cannot end simulation whenever some token arrives at the final destination. Instead, it must await a token with the desired tag.

To specify a constraint through a nondeterministic merge module then, we must specify the **tag** of the awaited final token. In general, each tag will consist of a partially ordered set. Each element within the set specifies: a nondeterministic merge module, the set of tagged input tokens which invoked that module, and the shared output stream.

The specification of the tool recovery constraint, for instance, becomes:

$$s_2[0,150] \wedge s_{18}[0] \quad \rightarrow \quad s_{1Tool}[300\{\texttt{Acquisition}: s_{17} \rightarrow s_1\}].$$

and

MonitorMerge	state 1:	$s_{12} \wedge s_{14}$	$\rightarrow$	s_{17}
Module	next state:	state 2		
Timing				
Specification	state 2:	$s_{12} \wedge s_{14}$	$\rightarrow$	s_{17}
			$\rightarrow$	
	next state:	state 2.		

Here, the constrained token along input stream 1 of the `Tool` module is tagged. Its tag specifies that the token was once merged onto a shared stream, stream 1, by the `Acquisition` module. That invocation consumed one token along input stream s_{17}.

In general, we can expand the specification of any simple constraint (Equation 5.6) to include tags:

$$\begin{aligned} & s_1[t_{1,1} \cdots t_{1,n_1}] \wedge s_2[t_{2,1} \cdots t_{2,n_2}] \wedge \cdots \\ & \quad \rightarrow \quad s'_{1M_1}[d_{1,1}\{tag_{1,1}\} \cdots d_{1,m_1}\{tag_{1,m_1}\}] \wedge s'_{2M_2}[d_{2,1}\{tag_{2,1}\} \cdots d_{2,m_2}\{tag_{2,m_2}\}] \wedge \cdots . \end{aligned} \quad (5.7)$$

5.4 Summary

In this section, we have specified real-time constraints through feedback processes. Using our simple simulator, we first showed how to verify a constraint by stepping through a simulation. Based on this interaction, we extracted information which defined the constraint. Figures 5.6 and 5.5 show the final specification of SAMPLE's feedback and tool recovery constraints.

$$s_2[0,150] \wedge s_{18}[0] \quad \rightarrow \quad s_{1_{Tool}}[300\{\texttt{Acquisition}: s_{17} \rightarrow s_1\}].$$

and

$$\begin{array}{lllll} \mathit{MonitorMerge} & \text{state 1:} & s_{12} \wedge s_{14} & \rightarrow & s_{17} \\ \mathit{Module} & \text{next state:} & \text{state 2} & & \\ \mathit{Timing} & & & & \\ \mathit{Specification} & \text{state 2:} & s_{12} \wedge s_{14} & \rightarrow & s_{17} \\ & & \rightarrow & & \\ & \text{next state:} & \text{state 2.} & & \end{array}$$

Figure 5.5: Final Specification of SAMPLE's Tool Recovery Constraint.

$$s_2[0] \wedge s_{18}[0] \quad \rightarrow \quad s_{1_{Tool}}[150\{\texttt{Acquisition}: s_{16} \rightarrow s_1\}].$$

and

$$\begin{array}{lllll} \mathit{MonitorMerge} & \text{state 1:} & s_{12} \wedge s_{14} & \rightarrow & \\ \mathit{Module} & \text{next state:} & \text{state 2} & & \\ \mathit{Timing} & & & & \\ \mathit{Specification} & \text{state 2:} & s_{12} \wedge s_{14} & \rightarrow & s_{17} \\ & & \rightarrow & & \\ & \text{next state:} & \text{state 2.} & & \end{array}$$

Figure 5.6: Final Specification of SAMPLE's Feedback Constraint.

With our constraint specification, we are now ready to verify contention free constraints. We are ready to take program's constraint specification together with program's timing specification and verify that constraint. In the next Chapter, we will extend our simulator of Section 4.3 into a constraint verifier and explore in greater detail the design decisions behind this verifier.

Chapter 6

Verification

Having finished specification, we are ready to tackle our last goal - verification of constraints. In the previous chapter, we have used the simulator of Section 4.3 to informally verify constraints. We will first extend this simulator to accept and to verify constraints. We will then discuss issues which we faced in our more realistic implementation. We will reevaluate the need for an event-driven verification and conclude with a description of a tagging scheme which we used to avoid duplication of work by multiple verifications.

6.1 Verification

We need to make several minor changes to the structure and control flow of our simulator in order to verify constraints. First, our verifier must accept a constraint specification and initialize its state accordingly. Second, the verifier must tag tokens as they pass through nondeterministic merge modules in order to identify their path. And finally, each time a token arrives, the verifier must check whether the arrival satisfies a constraint.

6.1.1 Static Description

The simulator was initially presented with a static description of the program. This description embodied all the information which we have accumulated about a program and its allocation. We now extend this description to include constraints. The verifier is presented with a formal specification of the constraint it is to verify (Chapter 5).

6.1.2 Runtime State

The verifier also needs two additional runtime structures to simulate the passage of time and to accumulate results. First, at initialization, the verifier must create a place to record the arrival times of constrained final tokens. As the verification begins, the arrival times of constrained tokens are empty. As the verification progresses, the arrival times slowly fill in.

Second, the verifier must append the description of each token with a tag. Each token must carry a tag describing its path through nondeterministic merge modules (Section 5.3). Each time a module fires, its output tokens must inherit the tags of all input tokens. In addition, each time a nondeterministic merge module fires, its output tokens must extend their tag as we saw in Section 5.3.

6.1.3 Verification

The overall control flow of the verification is somewhat more complex than that of the simulator. The verification first initializes the verifier based on the constraint. It then loops handling the next event. Whenever a new token arrives, the verifier checks whether the constraint has been met. The following sections describe the two new components of this verification: initialization and constraint testing, together with any changes to simulator's individual event handlers.

Initialization

To initialize a verification, we create places to record the arrival times of constrained final tokens. We replace the specifications of redefined modules. And finally, we enqueue create-token events for all initial tokens.

For instance, consider initialization of the feedback constraint verification. Figure 5.6 gave a formal specification of the feedback constraint. First, we create a place to record the arrival time of one token along stream 1 at module `Tool` whose tag is $\mathit{Acquisition} : s_{16} \rightarrow s_1$.

Next, we replace the MonitorMerge module's original specification:

$$
\begin{array}{rcl}
s_{12} \wedge s_{14} & \rightarrow & s_{17} \\
 & \rightarrow &
\end{array}
$$

with

$$
\begin{array}{lccc}
\text{state 1:} & s_{12} \wedge s_{14} & \rightarrow & s_{17} \\
\text{next state:} & \text{state 2} & & \\
 & & & \\
\text{state 2:} & s_{12} \wedge s_{14} & \rightarrow & s_{17} \\
 & & \rightarrow & \\
\text{next state:} & \text{state 2.} & &
\end{array}
$$

Finally, we enqueue two events to create feedback constraint's two initial tokens - one along stream s_2 at time 0 and another along stream s_{18} also at time 0. The type of these events is create-token signifying that a token is to be created at time 0 along the specified streams. These two events form the initial event queue in Figure 4.8:

```
0 create-token s2
0 create-token s18
```

Constraint Testing

The original simulator stepped through time indefinitely, simulating the timing behavior of a program. Our verifier, on the other hand, should terminate when all of constraint's final tokens have arrived. Each time a token arrives at its destination, the verifier checks whether it is a final token. In terms of our three event handlers - token creation, token arrival, and invocation completion - the verifier extends the token arrival handler to do this check.

The new token arrival handler compares the newly arrived token against constraint's final tokens which have not arrived yet. It looks for final tokens along the same stream, with the same destination module, and with the same tag. If the event handler does find such a final token, it records its arrival time.

Take, for instance, the feedback constraint verification. In one fork of this verification, a token arrival event is scheduled at time 80 along stream 1 at module `Tool` with tag *Acquisition* : $s_{16} \rightarrow s_1$. The token arrival handler compares this stream, destination module, and tag against that of constraint's final token and finds the two match. It records the arrival time of this final token to be 80.

As the verification progresses, the arrival times of constraint's final tokens fill in. Once all arrival times have been filled, the verifier terminates the simulation and checks whether all deadlines exceed arrival times. In case of the feedback constraint, both forks of the verifier terminate successfully, having met the imposed deadline.

6.1.4 Implementation Summary

In this section, we have extended the simulator of Section 4.3 in order to verify constraints. We have initialized our simulation based on its constraint, tagged tokens through nondeterministic merge modules, and terminated each simulation once all final tokens arrived. Clearly, the verifier we outlined in this section is neither efficient nor adequate. In the next section, we will discuss issues of interest in our design of a realistic verifier.

6.2 Extensions and Issues

The simple verifier outlined in the previous section leaves much to be desired. For one, it does not handle resource contention. In this section, we will outline further extensions to the verifier. And justify some of the decisions made in our implementation.

6.2.1 Contention

Our verifier of Section 4.3 ignored most contention costs. The verifier acknowledged that several potential invocations could contend for one module. However, once running a module invocation was guaranteed to complete without interruption. Our attempted justification for this guarantee pointed to non-overlapping allocation. We allocated each stream to a separate channel and and each module to a separate processor.

Unfortunately, even with non-overlapping allocation, a module invocation is not guaranteed to run without interruption. Each arriving token may have to interrupt the processor and raise module's completion time.

Moreover, a realistic verifier will need to handle overlapping allocations. SAMPLE's allocation is an example (Figure 3.18). With multiple allocations, each processor and each channel must be treated as a resource with waiting queues and general allocation methods. Additional events must handle processor and channel deallocation.

6.2.2 Initial Tokens

Our timing constraint specification from Chapter 5 gave no guideline as to what our set of initial tokens should be. It is important to realize that we cannot simply drop those initial tokens which, through program's behavior, will eventually generate final constrained tokens.

Any tokens that could affect the *timing* of final constrained tokens must be considered. Other tokens could affect timing through contention for shared resources or shared modules.

In general, finding a sufficient set of initial tokens must be left to the user. For periodic programs however, the set of periodically injected initial tokens could be defined as part of the program's behavior specification. We leave this definition for further work.

6.2.3 Regeneration

In the previous chapter, we implicitly assumed a single regeneration stage given one set of constrained input tokens. In other words, we assumed that the propagation of a single set of constraint's initial tokens through the constrained program defined the steady state of that program. It followed that the state of the program before propagation was equivalent to the one following the propagation. Under this assumption, a single propagation of constraint's initial tokens was representative of all, since all propagations were identical.

This assumption can be easily checked and relaxed. Following a propagation, we check for any unconsumed tokens within the program graph. For periodic programs with complete set of initial tokens, the only unconsumed tokens should be initialization tokens, such as the token along SAMPLE's acknowledgment stream[1]. If any other tokens remain then the verification should be repeated until a steady state is reached.

An example of a constraint which must be repeated is SAMPLE's feedback constraint:

$$s_2[0] \wedge s_{18}[0] \quad \rightarrow \quad s_{1_{Tool}}[300\{\texttt{Acquisition}: s_{16} \rightarrow s_1\}].$$

One propagation of this constraint's input tokens leaves an unconsumed token along stream 12. Within a second propagation, the `MonitorMerge` module consumes this token together with one along stream 14. The feedback constraint is met only if both propagations meet the imposed deadline.

6.2.4 Constant Latency

Our simple verifier substituted the maximum possible latency for each latency range. This substitution was not realistic. In addition to data dependent ranges of Table 4.3, parameters such as broadcast channel contention point to latency distributions rather than latency constants. The verifier assumed that by using the maximum possible latencies along the way, it

[1]It might be useful to specially identify initialization tokens in constraint specification.

would find the maximum possible latency through the entire constrained path. Unfortunately, this assumption breaks down in the presence of contention for shared resources or for shared modules.

We illustrates the fallacy of this assumption on the simple program of Figure 6.1. The timing behavior of the three modules in Figure 6.1 is:

$$\begin{array}{lccl}
\text{M1 behavior:} & s_1 & \overset{3}{\rightarrow} & s_3[3] \\
\text{M2 behavior:} & s_2 & \overset{2-4}{\rightarrow} & s_4[2-4] \\
\text{M3 behavior:} & s_3 & \overset{10}{\rightarrow} & s_5[10] \\
 & s_4 & \overset{10}{\rightarrow} & s_5[10]
\end{array}$$

Assume, for simplicity, that all channels have zero latency and each module is allocated to a separate processor. The imposed constraint is:

$$s_1[0] \wedge s_2[0] \quad \rightarrow \quad s_{5\,M\,destination}[15\{\mathtt{M3}: s_3 \rightarrow s_5\}].$$

Figures 6.1 and 6.2 show the inadequacy of verification based on maximum latencies. Using the maximum latencies along modules, we find that the constrained final token arrives at time 13, well within the deadline. Using the minimum latencies instead, we see that the constrained final token will not arrive until time 22.

One solution would be for our verifier to fork one verification for each permutation of overlapping ranges. This could be very costly. A much less computationally intensive solution is to enforce as many maximum latencies as possible at runtime. Whether we need to or not, we promise to wait until the maximum latency. In case of module M_2, for instance, we promise to generate a token and terminate at time 4. This is the approach we adopt in this chapter. A more desirable solution would be to find a bound on the introduced deviation from maximum latency.

6.3 Alternative to Event-Driven Verification

Our verifier was event driven. The advantage of this approach was accurate simulation of state at each point in time. For each decision, the simulator presented an accurate snapshot of computation state at the time of the decision. The disadvantage of this approach was the cost. To step through events sequentially in time, we had to maintain and sort an event queue.

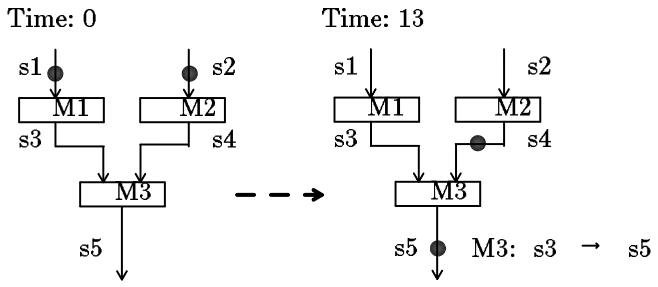

Figure 6.1: Verification Using Maximum Latencies.

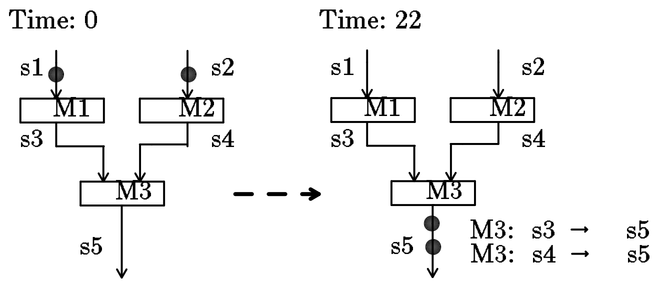

Figure 6.2: Verification Using Minimum Latencies.

An alternate approach would be to eliminate the event queue. Instead of queueing each new event and handling events in order, we could handle events recursively - regardless of their time. We will refer to this modified verifier as the unordered verifier. For instance, the new token creation handler would not return control to a main loop. Instead of enqueueing an arrive-token event at a later time, it would invoke the token arrival handler right away. Control would return to the top level only when a sequence of events terminated.

Figure 6.3 illustrates on a verification of the feedback constraint. It shows the order in which events are handled by our modified verifier. We see that events up to time 149 are handled before an event at time 0. Surprisingly, the unordered handling of events did not effect the correctness of this verification. All events and their times are identical to those of Figure 4.8.

This is not always the case. In the next few sections, we will look at ways in which the absence of event ordering could compromise the correctness of our verification. Not surprisingly, we will find that any source of nondeterminism in our programming model cannot be simulated correctly by an unordered verifier. In addition, contention costs can, at best, be bound.

0 create-token s2
↓
4 arrive-token s2 Acquisition
↓
9 complete-invocation Acquisition
↓
9 create-token s3
↓
19 arrive-token s3 ModelSelect
↓
9 arrive-token s3 MonitorSelect
↓
19 complete-invocation MonitorSelect
↓
19 create-token s11
↓
19 arrive-token s11 Monitor1
↓
149 complete-invocation Monitor1
↓
109 create-token s12
↓
109 arrive-token s12 MonitorMerge
↓
0 create-token s18
↓
...

Figure 6.3: Part of a Unordered Verification of the Feedback Constraint.

6.3.1 Nondeterminism

First, we look at ways in which nondeterminism in our programming model could compromise the correctness of our verification. There are three sources of nondeterminism in Stream Machine programs. All three originate from nondeterministic merge modules and their extension, the generalized merge modules (Section 4.1). In terms of our abstract behavior specification, all three originate from the presence of multiple statements. Next, we examine each of the three sources of nondeterminism and consider how the presence of each affects the correctness of an unordered verification.

Multiple Statements

The first source of nondeterminism is caused simply by the presence of **multiple statements**. Take the simplest set of two statements:

$$\begin{aligned} s_1 &\rightarrow s_3 \\ s_2 &\rightarrow s_4 \end{aligned}$$

The order in which individual statements are invoked depends on the order in which tokens arrive along input streams. That is, the order of invocations is time dependent or nondeterministic.

Time dependent order of invocations compromises the behavior of multi-state modules in an unordered verification. As later input tokens are handled first, they may fire the wrong statement first and wrongly propel the module into its next state.

Consider a module with the following behavior:

$$\begin{array}{lrcl} state: 1 & s_1 & \rightarrow & s_3 \\ & s_2 & \rightarrow & s_3 \\ nextstate: 2 & & & \\ state: 2 & s_2 & \rightarrow & s_4 \\ nextstate: 2 & & & \end{array}$$

Say that one token arrives along stream 1 and sometimes later one token arrives along stream 2. Figure 6.4 illustrates. From the module's timing behavior, the arrival of a token along stream 1 will fire statement $s_1 \rightarrow s_3$ and transfer the module into state 2. Later in state 2, the arrival of a token along stream 2 will fire the statement $s_2 \rightarrow s_4$.

Consider what happens if we handle the arrival of a token along stream 2 first (Figure 6.5). The arrival of a token along stream 2 will wrongly invoke the $s_2 \rightarrow s_3$ statement and wrongly transfer the module into state 2. Later on in state 2, a token will arrive along stream 1 and remain queued forever. Clearly, the module's behavior will be compromised.

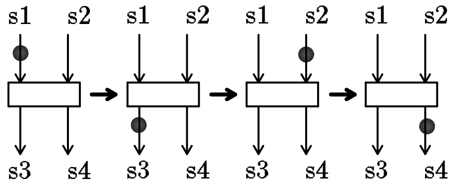

Figure 6.4: Correct Verification.

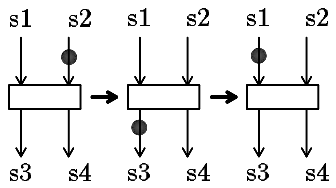

Figure 6.5: Unordered Verification.

Shared Output Stream

A second source of nondeterminism is caused by the sharing of a **common output stream** by several statements. The simplest merge module:

$$
\begin{aligned}
s_1 &\rightarrow s_3 \\
s_2 &\rightarrow s_3
\end{aligned}
$$

in Figure 3.4 illustrated. The order in which tokens are merged onto the output stream depends on the order in which tokens arrive along input streams `Left` and `Right`. The order of tokens along the output stream is therefore nondeterministic.

Two difficulties arise as a result of misordered handling of tokens along shared output streams. The handling of tagged tokens and the timing of successive modules' invocations are compromised.

Tagging First, the behavior of tagged constraints is compromised. To illustrate, consider the program:

$$
\begin{array}{lrcl}
\text{M1 behavior:} & s_1 & \stackrel{100}{\rightarrow} & s_3[100] \\
\text{M2 behavior:} & s_2 & \stackrel{10}{\rightarrow} & s_4[10] \\
\text{M3 behavior:} & s_3 & \stackrel{10}{\rightarrow} & s_5[10] \\
 & s_4 & \stackrel{10}{\rightarrow} & s_5[10] \\
\text{M4 behavior:} & s_5 \wedge s_6 & \stackrel{10}{\rightarrow} & s_7[10] \\
\text{M5 behavior:} & s_7 & \stackrel{10}{\rightarrow} &
\end{array}
$$

The imposed constraint is:

$$s_6[0] \wedge s_1[0] \wedge s_2[0] \quad \rightarrow \quad s_{7M5}[20\{\texttt{M3} : s_4 \rightarrow s_5\}].$$

Figure 6.6 shows the progress of this verification. Since tokens arrive at stream 5 unordered, the invocation of module M4 may consume the wrong token. As a result, the imposed constraint will go unmet.

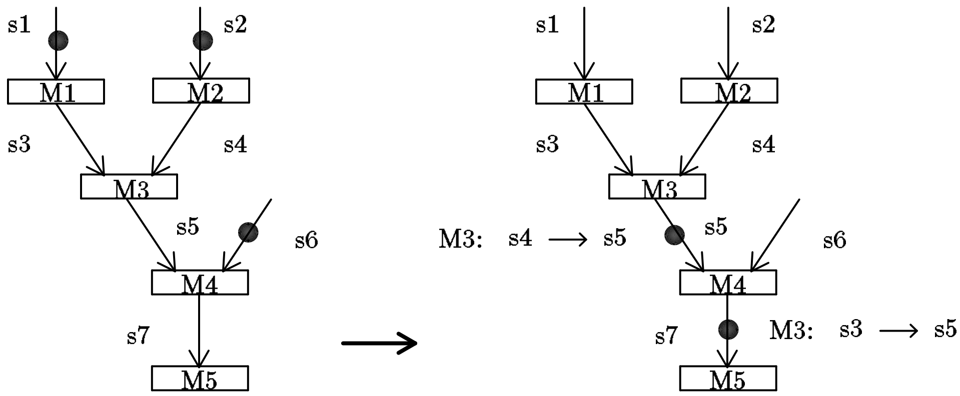

Figure 6.6: Unordered Verification.

Invocation Times Even if constrained final tokens do not contain tags, the timing of our program may be compromised. This is because module invocations may consume wrong tokens and, as a result, fire at the wrong time. Figure 6.6 illustrated this. Module M4 consumed the wrong token along stream 5 and, as a result, fired at time 110 instead of 20. It is important to notice that invocation times in our unordered verification are guaranteed to equal or exceed the correct invocation times. Constraints will not be compromised because of these incorrect invocation times, although they may go needlessly unmet.

Overlapping Input Streams

A final source of nondeterminism is caused by the sharing of a **common input stream** by several statements. Take the case:

$$\begin{aligned} s_1 \wedge s_2 &\rightarrow s_3 \\ s_1 &\rightarrow s_4 \end{aligned}$$

Depending on the time at which an input token arrives at stream 1, its arrival can complete one or both input sets. The number of input sets it completes is nondeterministic.

The time dependent number of completed input sets can cause incorrect behavior. To illustrate, consider the module:

$$\begin{aligned} s_1 \wedge s_2 &\rightarrow s_3 \\ s_2 &\rightarrow s_3 \end{aligned}$$

As before, say one token arrives along stream 1 and sometimes later one token arrives along stream 2. If we handle arriving tokens in order, the first token will queue onto the input queue of stream 1. The second token along stream 2 will complete both input sets. Which input set will actually fire is nondeterministic and our verifier will fork a separate verification for each case (Figure 6.7).

If we handle arrived tokens out of order, the module will behave incorrectly (Figure 6.8). A token along stream 2 will complete only one input set and fire. Later on a token will arrive and queue indefinitely onto the input queue of stream 1.

Summary of Nondeterminism in Unordered Verification

In this section, we have identified modules for which an unordered verification is not feasible. All such modules behave nondeterministically. As events are handled out of order, their nondeterministic behavior is compromised.

6.3.2 Contention

An additional phenomenon compromises the correctness of an unordered verification - contention. In case of module contention, the verifier must be able to tell whether another invocation is occupying a module. In case of resource contention, the verifier must be able to tell

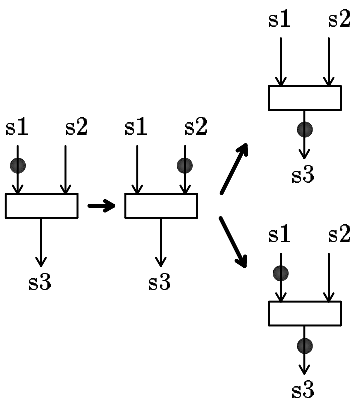

Figure 6.7: Correct Verification.

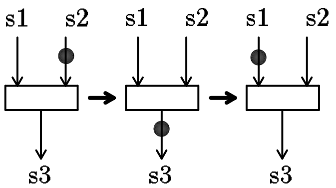

Figure 6.8: Unordered Verification.

whether another consumer is occupying the resource. But as events are handled unordered, the verifier does not have an accurate snapshot of the program's current state.

Module Contention

Our programming model dictates that each module have at most one invocation at any given time. An event driven verifier maintained this constraint by setting the current allocation variable while a module was invoked. An unordered verifier can maintain the same constraint by recording the most recent completion time for each module. If another invocation fires before the completion time, its invocation latency will be increased by the remaining completion time of the previous invocation. This procedure will guarantee at most one invocation of each module at a time.

But it will not guarantee the correct order of invocations. As we have seen, the presence of nondeterministic modules will compromise the invocation order. Figure 6.6 also illustrates this. Since the arrival of tokens along streams 3 and 4 will be handled out of order, the two invocations of module M4 will be switched. The invocation $s_4 \rightarrow s_5$ will incorrectly have to wait for the invocation $s_3 \rightarrow s_5$ to complete.

Resource Contention

The order and time of consumers' allocations will not be preserved. In our event driven verifier, each consumer must compete with all other consumers at that time for a resource. To allocate a processor or a channel, we must know exactly which consumers have arrived. Without this knowledge, we will not be able to compute the correct time when a consumer will acquire its resource.

This problem is more serious than that of module contention. In the absence of nondeterminism, we could compute the time it takes an invocation to acquire a module. But we cannot compute the time it takes an invocation to acquire a processor.

Fortunately, in the absence of nondeterminism, we can compute a time which equals or exceeds the correct time to acquire a resource. To do that, we look at all consumers which will ever compete for a given resource. We accumulate these by simulating the program's behavior independent of time. For each consumer, we can then make the unrealistic assumption that it acquires the module last. This assumption is guaranteed to equal or exceed the correct invocation time[2].

Summary of Contention in Unordered Verification

To conclude, in the absence of nondeterministic modules, a unordered verifier can handle contention costs. It can compute the exact time it takes an invocation to acquire a module. And, by accumulating all consumers for each resource first, the verifier can place an upper bound on the time it takes a each consumer to acquire that resource. However, constraints met by the program may not be verified because of unrealistically high upper bounds.

6.3.3 Alternative Summary

In this section, we have looked at ways in which unordered event handling might produce incorrect results. We saw the presence of nondeterministic modules compromised the timing, and even behavior of an unordered verifier. Even in the absence nondeterministic modules, we saw that resource contention costs could only be bound.

[2] We can tighten the assumption by eliminating all consumers which are guaranteed to follow a given consumer.

6.4 Tagged Verification to Avoid Duplication of Verification

In this section, we address the overhead of multiple simulations. The event-driven verifier copied all runtime state (Section 4.3) each time it forked multiple verifications. We will look at an alternative way to manage multiple verifications.

As we saw in Section 4.3, the event-driven verifier forked multiple verifications in two places. Whenever the verifier fired a statement with multiple output sets, it forked one simulation for each output set. Each output set represented different timing behavior. To validate a constraint, each possible timing behavior had to be verified.

In addition, generalized merge modules could cause the verifier to fork multiple verifications. Say the arrival of one token completed several input sets. Which input set should consume the token? The programming model did not specify. Without an accurate knowledge of the Stream Machine implementation that is being verified, the verifier had to fork multiple verifications - one for each completed input set.

The forking of multiple verifications may lead to redundancy. Consider again our sample program with each module and each stream allocated to a separate processor or channel. Figure 4.6 showed the verifier forking two simulations as a result of `ModelSelect` module's multiple output sets:

$$\begin{aligned} s_3 \wedge s_{18} \quad &\rightarrow \quad s_4 \wedge s_8 \\ &\rightarrow \quad 2s_9 \wedge s_8 \end{aligned}$$

Figure 4.8 showed the outstanding events at each point in time. Notice that two identical events appear in each forked verification: the complete-invocation event and the create-token event for `MonitorSelect` and its output token along stream 11. As a result, each of the two verifications goes on to verify an identical monitor subgraph of SAMPLE.

This duplication of work arises because our verifier needlessly copies the entire runtime state when forking. As a result, events whose behavior and timing do not depend on the fork will be verified multiple times.

An alternate approach is to share state among multiple verifications. Figure 6.9 illustrates. Instead of duplicating the entire runtime state, the verifier tags each piece of runtime state. In Figure 6.9, tag *tag1* identifies all runtime state which is belongs to the $\rightarrow s_4 \wedge s_8$ branch. Tag *tag2* identifies all runtime state which belongs to the $\rightarrow 2s_9 \wedge s_8$ branch. Note that there are two distinct invocations of the `ModelSelect` module - one for each branch. On the other hand, a single invocation of the `Monitor1` module is shared by both branches.

Figure 6.10 shows the single shared event queue at each point in time up to the fork. Comparing Figure 6.10 to Figure 4.8, we see that the tagged verifier simply tagged all events when it forked two verifications. It tagged events unique to the $\rightarrow s_4 \wedge s_8$ branch with tag *tag1*, events unique to the $\rightarrow 2s_9 \wedge s_8$ branch with tag *tag2*, and events shared by both branches with both tags.

A natural question to ask is what does the next step of Figure 6.10 look like. Ignoring tags, we know how the simple verifier of Sections 4.3 and 6.1 would behave. But how do tags affect this behavior?

6.4.1 Tagged Verifier

In this section, we outline changes made to the simple verifier in order to handle tags.

Tagged State

As we have seen in Figure 6.10, the tagged verifier tags each piece of runtime state with the verifications to which that state belongs. Each token carries a tag identifying the verification(s) to which that token belongs. We will call this tag the verifications-tag to differentiate it from the tag described in Section 5.3. The tag described in Section 5.3 identifies the origin of a token which has passed through nondeterministic merge modules. It has no relation to the verification-tag.

All initial tokens start out as part of the same verification. As a result, they all share the same initial verification tag.

Each module carries a list of tagged current states. Since each verification has to be in some module state, each module carries as many tagged current states as there are verifications. Each module also carries a list of tagged current invocations - one for each verification which is currently invoking that module.

Finally, as we saw in Figure 6.10, each event carries a verification-tag identifying the verification(s) within which that event is pending.

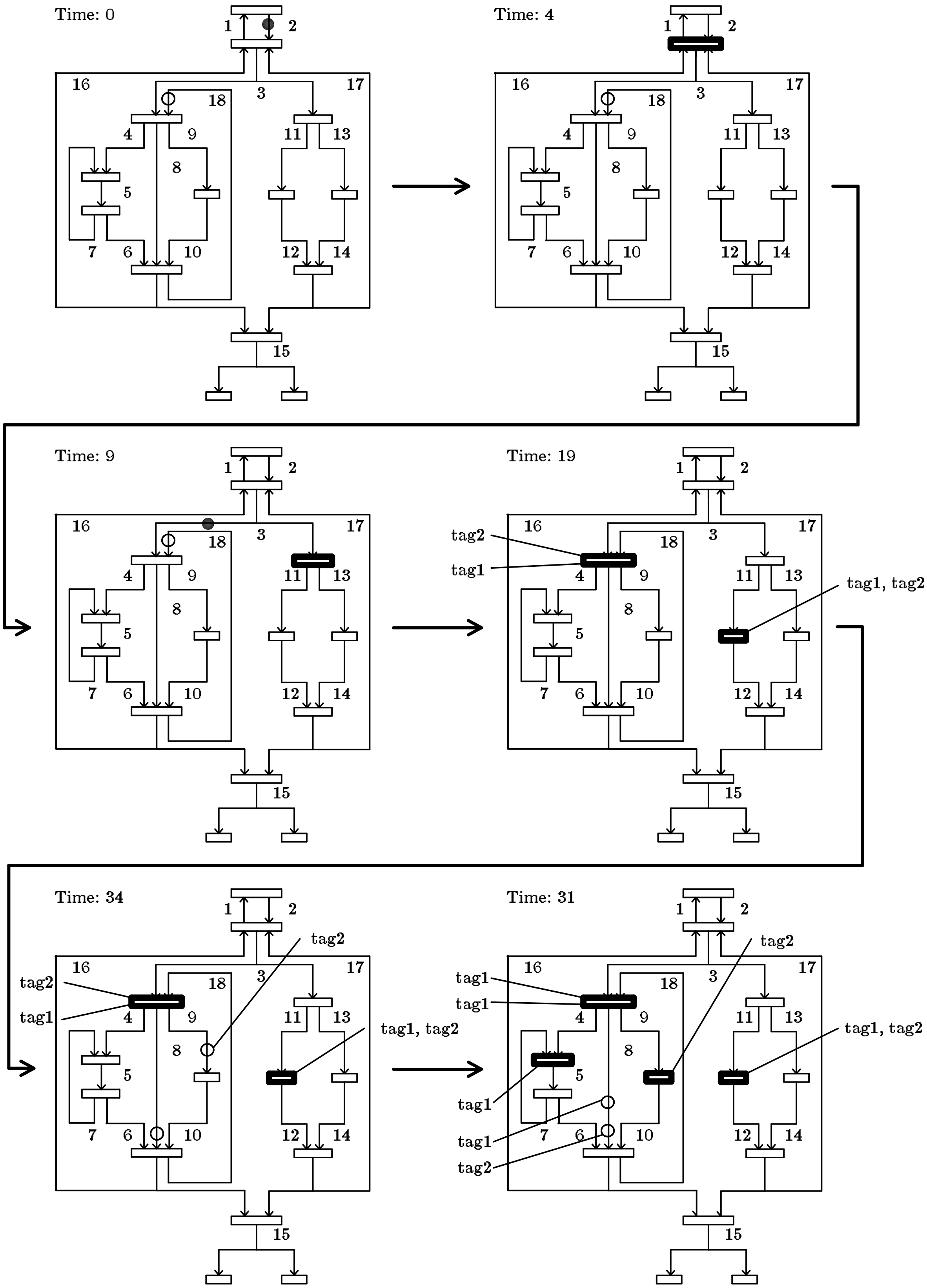

Figure 6.9: Part of Tagged Feedback Constraint Verification.

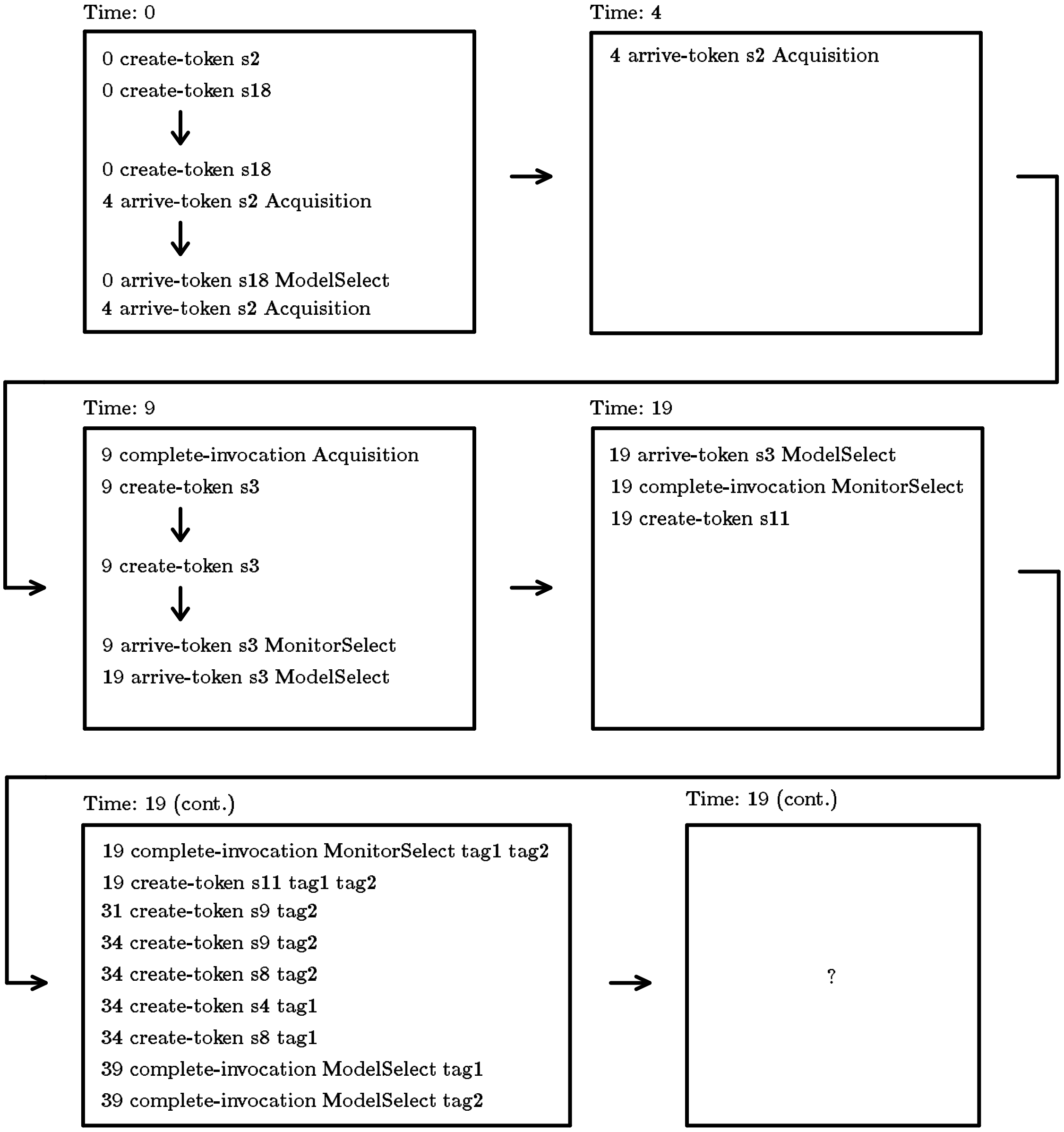

Figure 6.10: Events for Part of Tagged Feedback Constraint Verification.

Event Handling

Simple Events Each event is now part of some verification(s) specified by its verification tag. Consequently, each event should only consider runtime state which corresponds to its verification(s). Take the remaining events at time 19 after the fork in Figure 6.10 - the invocation completion of the `MonitorSelect` module, the creation of a token along its output stream 11, and eventually its arrival at the `Monitor1` module. All three events are part of two verifications, the *tag1* verification and the *tag2* verification. The tagged event handler handles these events the same way as the individual untagged event handlers for verification *tag1* and for verification *tag2* would have. Figures 6.11 and 6.12 show the state of our verifier and of its event queue before after these three events - right after the fork at time 19 and at the end of time 19.

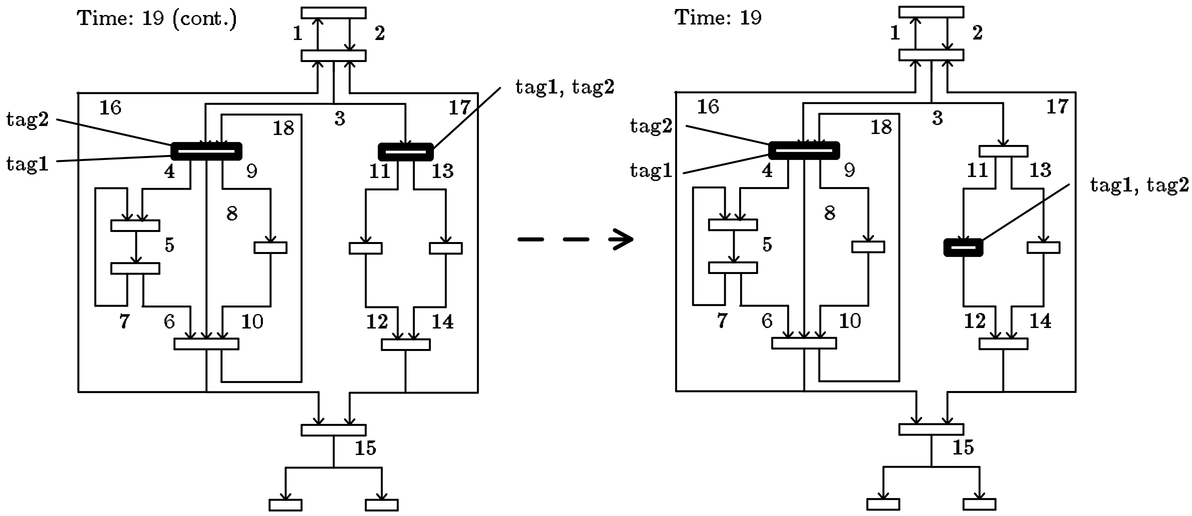

Figure 6.11: Part of Tagged Feedback Constraint Verification.

To complete invocation of the `MonitorSelect` module, the event handler empties the current invocation variable for verifications *tag1* and *tag2*. It also checks whether a new invocation is queued to run in verification *tag1* or in verification *tag2*. Since both verifications have an empty invocation queue, the event handler returns control to the main loop. The creation of a token along stream 11 triggers an 'arrive-token event. Since the token was created in verifications *tag1* and *tag2*, the token arrives at Monitor1 in verifications *tag1* and *tag2*. The arrival of a token along stream 11 invokes the `Monitor1` module in verifications *tag1* and *tag2*. This is the state of our verifier at time 31 in Figure 6.11.

A more illustrative point occurs at time 41. Figures 6.13 and 6.14 illustrate. A token arrives at the input stream 10 of the `ModelMerge` module in verification *tag2*. At this point two tokens

```
19 complete-invocation MonitorSelect tag1 tag2
19 create-token s11 tag1 tag2
31 create-token s9 tag2
34 create-token s9 tag2
34 create-token s8 tag2
34 create-token s4 tag1
34 create-token s8 tag1
39 complete-invocation ModelSelect tag1
39 complete-invocation ModelSelect tag2
```

```
31 create-token s9 tag2
34 create-token s9 tag2
34 create-token s8 tag2
34 create-token s4 tag1
34 create-token s8 tag1
39 complete-invocation ModelSelect tag1
39 complete-invocation ModelSelect tag2
109 create-token s12 tag1,tag2
149 complete-invocation Monitor1 tag1,tag2
```

Figure 6.12: Events for Part of Tagged Feedback Constraint Verification.

are queued at the input stream 8 - one in verification *tag1* and one in verification *tag2*. In verification *tag2*, a complete input set is present and an invocation of the `ModelMerge` module is fired. The input token along stream 8 in verification *tag1* stays intact.

Multiple Forks In the previous section, we have seen how tagged state is handled. But what happens when the tagged event handler encounters another fork?

This happens in the verification of the recovery constraint (Figure 5.4). At time 150 a second initial token is injected into stream 2. At time 169, the `ModelSelect` module forks for the second time. The module forks within two verifications - the *tag1* verification and the *tag2* verification. As a result, the forking event handler sees all state marked *tag1* or *tag2*. As was the case in the first fork, the event handler labels all (visible) state as belonging to one or both forks.

For instance, a final token along stream 1 was part of a single verification - *tag2*. After the fork, this token becomes part of two verifications - *tag2-tag3* and *tag2-tag4*.

In our example in (Figure 5.4), all state is visible to the forking event handler. In general, this need not be the case. Had a nested fork occurred inside verification *tag1*, all state inside

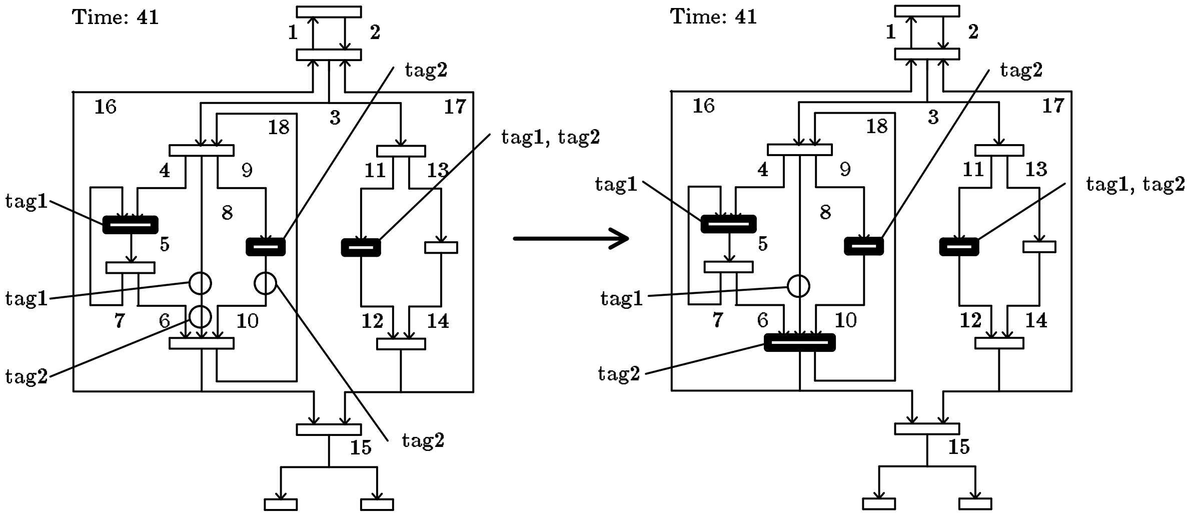

Figure 6.13: Part of Tagged Feedback Constraint Verification.

41 arrive-token s10 ModelMerge tag2
44 complete-invocation Sand tag2
44 complete-invocation Oil1 tag1
44 create-token s5 tag1
109 create-token s12 tag1,tag2
149 complete-invocation Monitor1 tag1,tag2

↓

44 complete-invocation Sand tag2
44 complete-invocation Oil1 tag1
44 create-token s5 tag1
61 complete-invocation ModelMerge tag2
61 create-token s16 tag2
109 create-token s12 tag1,tag2
149 complete-invocation Monitor1 tag1,tag2

Figure 6.14: Events for Part of Tagged Feedback Constraint Verification.

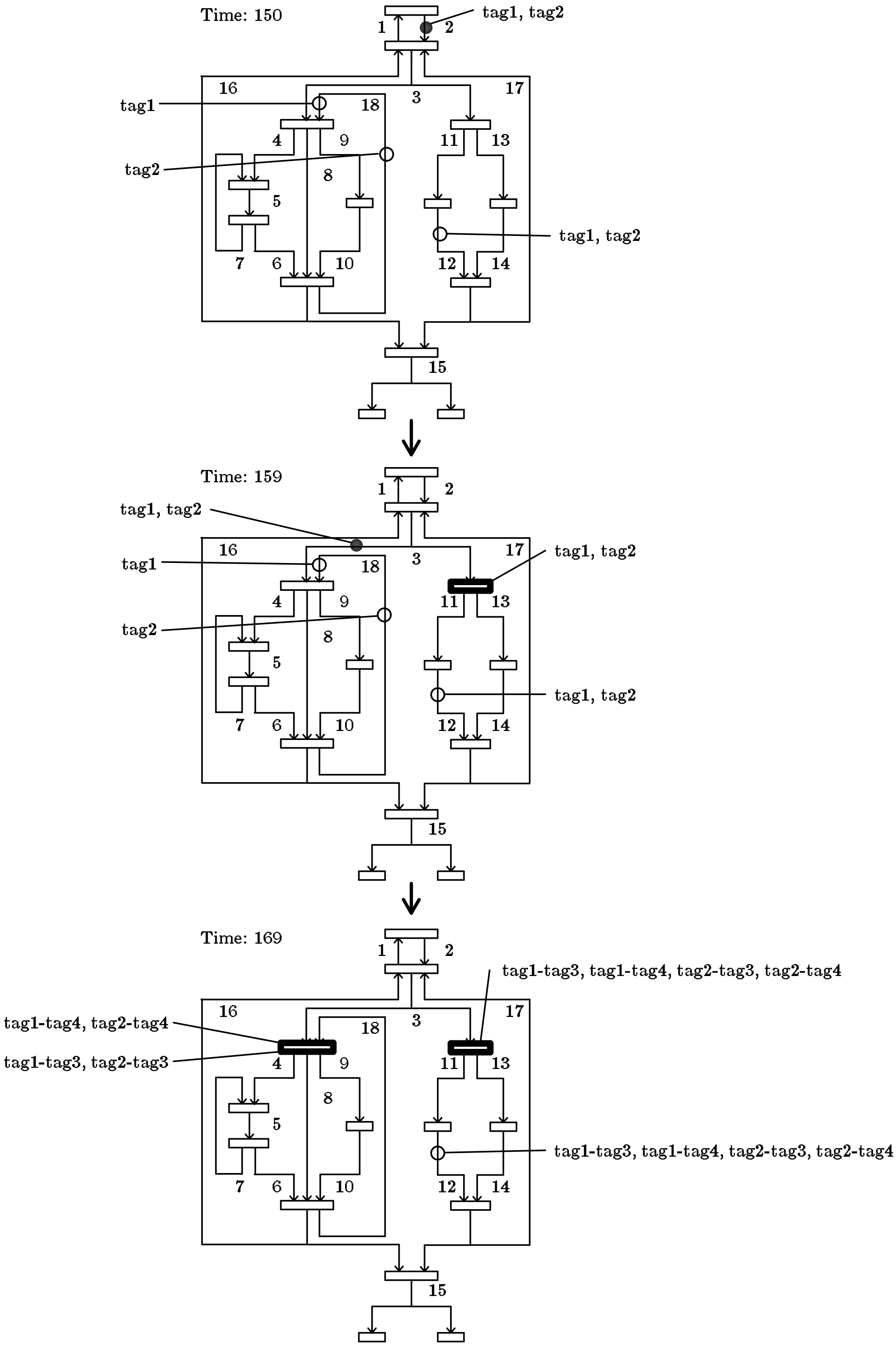

Figure 6.15: Part of Tagged Feedback Constraint Verification.

verification *tag2* would have remained unaffected. The final token along stream 1 marked *tag2* would have remained marked *tag2*.

6.4.2 Modified Tagged Verifier

Unfortunately, the tagged verifier described so far in this section does not reduce the cost of forking multiple verifications. Just like the simple verifier, the tagged verifier touches each piece of runtime state when forking. The simple verifier touched each piece in order to copy it. The tagged verifier touches each piece in order to extend its tag if necessary.

A modified version of the tagged verifier can significantly lower the cost of forking multiple verifications. The original tagged verifier tagged each piece of runtime state with the identity of all verification branches in which it existed. A modified version tags each piece of runtime state with the identity of all verification branches in which it does *not* exist.

This modification eliminates the need to touch each piece of runtime state when forking. Since each existing piece of runtime state will exist in both branches of the fork, its tag does not change. Only the immediate outcome of each branch must be tagged to indicate that it does not exist in its sibling branch.

6.4.3 Tagged Verification Summary

We have outlined the implementation of a tagged event-driven verifier. Unlike the simple event-driven verifier of Section 4.3, this verifier does not needlessly duplicate verifications at each fork. Instead, it tags each piece of runtime state as belonging (or in case of the modified version *not* belonging) to certain verification branches. As a result, the tagged verifier avoids verifying the same subgraph multiple times.

We have implemented the modified tagged verifier as part of this work. Further work is needed to evaluate its usefulness for various types of programs. The tagged verifier avoids the cost of copying runtime state and needlessly duplicating work at each fork. However, in return it incurs the cost of tag manipulation at each event.

Chapter 7

Conclusion

At the beginning of this project, we were presented with a distributed programming environment for control and data acquisition in the wireline industry. We were told of a need to guarantee a specific program response within a strict time bound. We narrowed our problem by concentrating on a specific program allocation.

One possible way to guarantee such response is by running the program for a set of input values which cover all possible response times. We rejected this approach for several reasons. First, our domain might not guarantee repeatable response times. Certain latencies might vary within a range. Second, access to the exact runtime environment - a wireline truck - is limited during the program development stage. Third, the control flow of our programs is expected to reach complexity level at which selection of sufficient input values is no trivial. And fourth, the expected granularity of our timing costs is high enough to make simulation feasible.

As a result, we opted for simulation. We clearified our notion of a program to a point where we could simulate all possible timing costs. And we clarified our notion of a program response to a point where we could verify whether a particular program response will take place within a strict time bound. As with most projects, much remains to be accomplished. The following two sections offer our ideas for improvements and future directions.

7.1 Improvements

We see two major areas for improvement, both in our program specification.

7.1.1 Linking Behavior of Modules

To achieve data independence, our module behavior specification abstracted away values along tokens. We could no longer tell which values along input tokens produced each set of output tokens. Similarly, we could not tell which values along input tokens lead to each possible latency of an output token or an invocation. At the level of a module, this abstraction succeeded in describing all possible timings and behaviors of that module.

However, at the program level this abstraction removed useful information. Our execution model was unaware of any correlation between modules' timings and behaviors. It simulated every, potentially infeasible, permutation of modules' possible timings and behaviors. As a result, a constraint could be falsely rejected if some infeasible permutation of modules' possible timings and behaviors could not meet the constraint.

Any future work should specify alignment between individual modules' timings and behaviors. As we suggested in our summary to Section 4.1, users could enforce such alignment through tags.

7.1.2 Specification of Periodic Input

We relied on constraint specification to describe the initial state of a verification through an input set. As we discussed in Section 6.2.2, selection of a sufficient initial set is affected by contention costs and so depends on a particular allocation. In retrospect, a cleaner approach would have been to specify periodic input as part of modules' behavior and timing specification.

7.2 Future Directions

We focused our interest in this project on a single known program allocation. A natural expansion of our work would be the development of an allocator. Our verifier could be utilized by a heuristic driven allocator. Development of proper heuristics which take into account real-time constraints in our domain is an important field for further research.

Bibliography

[1] John A. Stankovic. "Misconceptions About Real-Time Computing." *IEEE Computer,* pp10-19, October, 1988.

[2] E. G. Coffman. "Computer and Job-Shop Scheduling Theory." *John Wiley & Sons,* New York, 1976.

[3] E. Horowitz, S. Sahni. 'Exact and approximate algorithms for scheduling nonidentical processors." *Journal of ACM,* 23, April 1976, 317-327.

[4] Stephen A. Ward. "An Approach to Real Time Computation." *Proceedings of the 7th Texas Conference on Computer Systems,* pp26-34, 1978.

[5] Dennis W. Leinbaugh, Mohamad-Reza Yamini. "Guaranteed Response Times in a Distributed Hard-Real-Time Environment." *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 12, December 1986.

[6] Michael F. Coulas, Glenn H. MacEwen, Genevieve Marquis. "RNet: A Hard Real-Time Distributed Programming System." *IEEE Transactions on Computers,* Vol. C-36, No. 8, August 1987.

[7] Aloysius Ka-Lau Mok. "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment." *MIT LCS Technical Report 297,* Department of Computer Science, MIT, May, 1983.

[8] Farnam Jahanian, Aloysius Ka-Lau Mok. "Safety Analysis of Timing Properties in Real-Time Systems." *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 9, September 1986.

[9] M. Garey, D. Johson. "Computers and Intractability: a Guide to the Theory of NP-Completeness." *W. H. Freeman,* San Francisco, California, 1979.

[10] Glenn H. MacEwen, David B. Skillicon. "Using Higher-Order Logic for Modular Specification of Real-Time Distributed Systems." *Proceedings of a Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems,* Lecture Notes in Computer Science 331, Warwick, UK, September 22-23, 1988.

[11] F. Jahanian, A. K. Mok. "Safety Analysis of Timing Properties in Real-Time Systems." *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 9, pp890-904, September, 1986.

[12] J. L. Bergerand, P. Caspi, D. Pilaud, N. Halbwachs, E. Pilaud. "Outline of a Real Time Data Flow Language." *Proceedings of the Real-Time Systems Symposium,* San Diego, California, December 3-6, 1985.

[13] H. Wupper, J. Vytopil. "A Specification Language for Reliable Real-time Systems." *Proceedings of a Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems,* Lecture Notes in Computer Science 331, Warwick, UK, September 22-23, 1988.

[14] J. S. Ostroff, W. M. Wonham. "Modelling, Specifying and Verifying Real-time Embedded Computer Systems." *Proceedings of the Real-Time Systems Symposium,* San Jose, California, December 1-3, 1987.

[15] David R. Barstow, Paul S. Barth. "The Resource Allocation Problem for Stream Machine Programs." *Schlumberger Doll Research, 1988.*

[16] David R. Barstow, Paul S. Barth, Richard Dinitz. "SPHINX: A Programming Environment for Device Control Software." *Schlumberger Doll Research, 1987.*

[17] Paul S. Barth, Scott B. Guthery, David R. Barstow. "The Stream Machine: A Data Flow Architecture for Real-Time Applications." *1985 Eighth International Conference on Software Engineering.*

[18] David R. Barstow, Greenspan. "Using a Device Model as Domain Knowledge in the Automatic Programming of Software to Control Remote Devices." *Schlumberger Doll Research, 1986.*

[19] C. A. R. Hoare. "Communicating Sequential Processes." *Communications of the ACM,* 8, pp666-677, August, 1978.

[20] Paul R. Kosinsky. "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs." *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages.*

[21] J. Dean Brock, William B. Ackerman. "Scenarios: A Model of Non-determinate Computation." *Computation Structures Group Memo 206,* Laboratory for Computer Science, MIT, February 1981.

[22] Domenico Ferrari. "Computer Systems Performance Evaluation." *Prentice Hall,* New Jersey, 1978.

[23] John Henize. "Understanding Real-Time UNIX." *Concurrent Computer Corporation,* Westford, MA.

[24] Randal E. Bryant . "Simulation of Packet Communication Architecture Computer Systems." *MIT LCS Technical Report 188,* Department of Computer Science, MIT, November, 1977.

[25] David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent. "Measured Capacity of an Ethernet: Myths and Reality." *Proceedings of SIGCOMM '88,* ACM SIGCOMM, August, 1988.

[26] Robert M. Metcalfe, David R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the ACM,* 7, pp395-404, July, 1976.

[27] Steven L. Beuerman, Edward J. Coyle. "The Delay Characteristics of CSMA/CD Networks." *IEEE Transactions on Communications,* 5, pp553-563, May, 1988.