A Formal Model For Architecture-Independent Parallel Software Engineering

David C. DiNucci
B.S., Portland State University, 1983

A dissertation submitted to the faculty
of the Oregon Graduate Institute
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

March, 1990

The dissertation "A Formal Model for Architecture-Independent Parallel Software Engineering" by David Carl DiNucci has been examined and approved by the following Examination Committee:

Robert G. Babb II
Thesis Advisor
Associate Professor

Richard B. Kieburtz
Professor, Department Head

Michael Wolfe
Associate Professor

Harry S. Jordan
Professor
University of Colorado at Boulder

Dedication

To Tamae, who did not deny her faith

in the most difficult of times

To my father, who demonstrated how to get the job done

To my mother, who showed me what it means to never quit

Acknowledgements

I would like to acknowledge the students, staff, and faculty of Oregon Graduate Institute for their support. In attempting to keep one foot in theory and another in practice, I have relied on their expertise and experience in both camps. I would especially like to thank Robbie Babb for keeping me apprised of the needs of the parallel processing community and Michael Wolfe for his support and example. I would also like to thank the other members of my committee for their assistance, Dick Hamlet for his excellent guidance in the early stages of this work, Steve Otto and Doug Pase for inspiring discussions, and the many others (including "the gang") whose friendship helped to make my stay at OGI an enjoyable experience.

Table of Contents

1. Algorithms for Parallel Architectures 2

2. Related Work 15

2.1. Introduction 15

2.2. Shared Memory and Message Passing 16

2.3. Parallelizing Compilers 17

2.4. Linda 18

2.5. Unity 19

2.6. Reactive Kernel 20

2.7. Specification Languages 21

2.8. Dataflow Languages 21

2.9. Coarse Grain Data Flow 22

2.10. Actors 22

2.11. Strand 23

2.12. Paralation 24

2.13. Synchronous Models 24

2.14. Historical Perspective 25

3. F-Nets 26

3.1. Introduction 26

3.2. Building the Model 27

3.2.1. Architecture-Independent Ether Model 27

3.2.2. Separability 32

3.2.3. Independence of Number of Processors 34

3.2.4. Sample Problem 36

3.3. Higher Level Characterization 38

3.3.1. Instruction Specifications 38

3.3.2. Ensuring That Signatures are Correct 40

3.3.3. Multiple Readers and Buffering in Ether 41

3.3.4. Instructions Performing the Same Operation 42

3.3.5. Final Version of Sample Problem 43

3.4. Final Notes 46

4. Axiomatic Semantics and Formal Results 47

4.1. Introduction 47

4.2. Syntax 48

4.3. Semantics 51

4.3.1. Form of an Execution Graph 52

4.3.2. Axioms Constraining Execution Graphs 53

4.4. Execution Graphs as Partial Orders 56

4.5. Execution Graphs as Computations 58

4.6. Tracing an Execution 60

4.7. Execution Graphs with Identical Logs are Isomorphic 64

4.8. Toward Proving $\hat{e}_x = \hat{e}_{x'}$ 67

4.9. Conclusions 70

4.10. Final Note on the Effects of Order (Size of Control Domain) 70

5. Comparison with Other Models 73

5.1. Unity 73

5.2. Petri Nets 74

5.3. CCS 76

5.4. Functional Models 77

5.5. Guarded Commands 77

5.6. Graphical Specification Languages 78

5.7. Imperative Sequential Programs 78

5.8. Conclusion 79

6. Implementation 80

6.1. Introduction 80

6.2. Definition of a Valid Implementation 80

6.3. A Generic Implementation 82

6.3.1. Concrete F-Nets 83

6.3.2. Abstraction of Concrete F-Nets 84

6.3.3. Concrete Execution Graphs and Their Abstraction 87

6.3.4. Concrete Implementation 87

6.3.4.1. Strategy 88

6.3.4.2. Validity of Implementation Strategy 91

6.3.4.3. Pseudo-code for the Generic Implementation 93

6.4. Optimizing Concrete Execution for Differing Architectures 98

6.4.1. Shared Memory 98

6.4.2. Message-Passing 100

6.4.3. Final Implementation Notes 103

7. Future Directions and Conclusions 105

7.1. Introduction 105

7.2. Extensions to F-Nets 105

7.3. Software Tools 107

7.3.1. Debugging/Monitoring Tools 107

7.3.2. Parallel Restructuring Tools 109

7.3.3. Real-time Programming 110

7.4. Parallel Architecture 111

7.5. Conclusion 113

Table of Illustrations

Figure 3.1 First Attempt at the Sample Problem 31

Figure 3.2 Second Attempt at Sample Problem 37

Figure 3.3 Final Diagram of Sample Problem 40

Figure 4.1 Node Labels for Execution Graph 53

Figure 4.2 One Execution Graph for the Sample F-Net 54

Figure 4.3 A New Sample F-Net 62

Figure 4.4. An Execution Graph for F-Net in Figure 4.3 63

Figure 4.5. Comparable Order-4 and Order-2 F-Nets 71

Figure 5.1. F-Nets Modeled as Petri Nets 75

Abstract

A Formal Model For Architecture-Independent

Parallel Software Engineering

David C. DiNucci, Ph. D.

Oregon Graduate Institute, 1991

Supervising Professor: Robert G. Babb II

In the absence of a unifying model to describe parallel algorithms, existing architectures have served as the models. The resulting algorithms, expressed as sets of sequential processes which communicate via shared memory or message passing, are non-portable, and the component processes cannot be implemented according to an input-output specification alone. Determining the set of computations represented by such an algorithm often requires no less than simulating their execution. This dissertation develops a model, F-Nets, for expressing parallel algorithms in a manner which avoids many of these difficulties. Both high- and low-latency communication are efficiently accomodated, and processes can be implemented in any deterministic language. The possible effects of each process is completely determined by the input-output mapping it implements. Computations are defined as partial orderings of these process executions, and algorithms are represented graphically as folded computations. A formal axiomatic semantics is provided for unfolding algorithms into computations, as is an operational semantics which is used to describe efficient implementations of the model on various architectures. Some final observations and predictions are made for future work based on the model.

CHAPTER 1

Algorithms for Parallel Architectures

1.1. Introduction

What is a parallel algorithm? Decades after the advent of parallel processors, this is still not a simple question to answer, or even understand. The question is usually not put so bluntly, or is accompanied by information about the intended target architecture: whether the target is shared-memory or message-passing or SIMD, the number of processors that it contains, specifics about the interconnection topology or memory hierarchy, and the cost of communication.

Yet, algorithms have traditionally been considered as being independent of architecture. The same algorithm can be considered as instructions to a human solving a problem on a scratch pad or black board, a Turing Machine accessing a tape and performing state transitions, or a uniprocessor accessing memory and executing instructions. The fact that a computation for all of these devices has the same form—a sequence—unifies the concept of algorithm and provides for them to be written and analyzed (to some extent) without knowing their target.

The goal of this work is to define a model of computation which can be efficiently implemented on different MIMD architectures, and one which provides a natural setting in which to describe parallel algorithms. In order to capture the generality of the word "algorithm", the programs expressed within this model should serve equally well as instructions to a room full of humans, a set of Turing Machines, or a parallel processor. The remainder of this chapter will further define the goals of this model by first describing the similarities and differences between MIMD architectures, then the requirements of a parallel algorithm and the factors which might make one representation better than another, and finally the role that sequential languages might play in a parallel setting.

1.2. MIMD Parallel Architectures

In this section, we provide a very simple formalism by which parallel architectures can be compared and contrasted. The terminology used is by no means standard: each term is defined as it is used. We then describe the differences in various architectures in terms of architectural implementation, physical characteristics, semantics, and number of processors.

1.2.1. Formalism

An MIMD parallel architecture consists of some number of *sequential processors* and a communication medium, called *ether*. Each sequential processor has the use of a computational unit capable of executing one instruction at a time, and each instruction terminates in finite time. Each processor possesses some amount of local state (e.g. registers and local memory) which is inaccessible by other processors and which can affect the behavior of instructions executing on it. Processors do not necessarily perform instructions at the same rate, nor do they have access to synchronized clocks. Each processor will be live—i.e. it will be ready to execute another instruction within a finite time after the last has completed.

Processors share access to the ether. In addition to the standard instructions performed by sequential processors, which only affect and are affected by the local state of the processor, each processor is capable of posting (performing) *events*. An event consists of a type, an address[1], and an optional data item. Event types are divided into primary-

[1]In a message-passing system, this address is often called a channel.

secondary pairs. Common event type pairs for current architectures are `(write, read)`, `(send, receive)`, and `(unlock, lock)`.

When a processor posts a primary event, that event is copied to the ether. When a processor posts a secondary event, the processor stalls until an associated primary event is found in the ether with the same address; if the primary event has an associated data item, that data is copied to the processor posting the secondary event. We call this pairing of primary and secondary events *event matching*.

1.2.2. Number of Computational Units

The number of computational units in MIMD parallel architectures vary widely. The definition given above does not preclude the use of a single computational unit by several processors[2]. In the general case, in order to preserve the liveness of each processor, the usage of the computational unit must be shared by interleaving groups of instruction executions from each processor. This requires that the computational unit switch contexts regularly—i.e. that the state of one processor be saved from the computational unit's state (e.g. registers, program counter, pointers to address space), and that the computational unit adopt the state of another processor. In general, an algorithm which is encoded to use many more processors than there are computational units will consume more time for context switching than one which is encoded to use approximately the same number of processors as computational units. An algorithm which uses fewer processors than there are computational units will use those units ineffectively.

This work will address this problem by presenting a program as a fairly large collection of fairly small segments. When a segment begins execution, it will have no state, nor

[2] In common usage, these processors would be called processes or virtual processors.

will it have state when it has completed. A segment will not begin execution until all of the resources that it needs are available to it, so a segment will never need to pause during its execution. In this way, a program execution consists of packing (i.e. scheduling) these segment executions onto the computational units at hand, with little or no need to switch contexts during segment executions, and no need to save or restore state between segment executions.

1.2.3. Architectural Implementation of Ether

Current architectural implementations of ether can be broadly categorized by the scalability of the pending event store and of the communication bandwidth required to match pending primary events with secondary events, where scalability refers to the ability of the architecture to accommodate more processors. Non-scalable communication is often implemented with a small number of fixed-bandwidth busses used for communication by all processing units, while scalable communication is implemented using a network of communication channels which grows with the number of processors. A non-scalable event store is often implemented by memory units which share common access paths, while a scalable event store is often implemented with several memory units, each with one or more autonomous access paths. Non-scalable event stores are often augmented with scalable caches.

1.2.4. Physical Characteristics of Ether

The physical characteristics of ether can be analyzed in terms of bandwidth, latency, and overhead. *Bandwidth* is the number of events and/or size of events that can be under transport at any one time. *Latency* is the time required to communicate (the news of) an event from one arbitrary processor to another. *Overhead* is the amount of processor time required to post an event.

These physical characteristics are related to architectural characteristics. Scalable communication is usually accompanied by relatively high latency, due in part to the fact that the larger number of processors accommodated requires longer communication distances. More significantly, the higher latency can be attributed to the techniques used to implement scalable communication. Since implementing a unique communication path from every processor to every part of the event store would be prohibitive, some paths are shared and/or pass through intervening processors, and routing data through these paths takes time. Scalable communication is also usually accompanied by high overhead: High latency makes it advantageous to associate a large amount of data with each data event, so the communication system must be prepared to handle these large, variable-length events.

1.2.5. Semantics of Ether

The semantic characteristics of the ether can be categorized into buffering, destructiveness, data (and granularity), and partitioning. An event pair is *buffered* if the ether can accommodate multiple primary events for a given address, unbuffered if it keeps only the latest. Even ether which supports buffered events is not infinite, and may lose events or fail catastrophically if too many unmatched primary events are posted. For buffered event pairs, if multiple primary events are found which match a secondary event, the oldest primary event is usually used for the match. An event pair is *destructive* if the secondary event removes the matched primary event from the ether, non-destructive if the primary event is not removed. An event pair is a *data* pair if the primary event contains a data field which is copied to the processor posting the secondary event when a match occurs. A data pair will be said to have large *granularity* if the data associated with the event can be large and variable-sized, fine granularity if the data is small and fixed size. An event pair is *partitioned* if the address associated with the primary event uniquely identifies the processor which will post the secondary event.

These semantic characteristics are traditionally a result of physical characteristics. As already mentioned, in a high-latency ether, it is advantageous to use large-granularity events. The use of partitioning can reduce the effect of latency by allowing the primary event to be forwarded directly to the processor which will post the secondary event, thereby avoiding the latency when the secondary event is posted. Buffering allows latency to be hidden through pipelining, and allows multiple primary events to be posted to a given address without waiting for verification through the ether (i.e. handshaking) that the previous primary events have been matched each time. For these reasons, a popular semantic combination for high-latency ether consists of a `(send, receive)` large-granularity event pair which is buffered, destructive, and partitioned. We call this combination of semantics *message-passing*.

When the latency of the ether is low, the amount of data passed in a data event can be small, and to keep overhead low, a fixed size datum is usually used. Partitioning is not needed, and is restrictive: if the processor which will post the secondary event is determined without knowledge of the processor which will post the primary event, partitioning adds extra overhead by requiring an extra event match to communicate this "demand" to the primary event poster. Buffering is also not needed, and imposes extra overhead. For these reasons, a popular semantics for low-latency ether consists of a `(write, read)` fine-granularity event pair which is unbuffered, non-destructive, and non-partitioned, and a `(relinquish, acquire)` no-data event pair which is unbuffered, destructive, and non-partitioned. We call this combination of semantics *shared-memory*.

The use of destructiveness to block a processor from taking action until it is safe (i.e. to achieve synchronization) is independent of latency, and so is present in both shared-memory and message-passing semantics. It plays the additional role of removing events from the ether in message-passing semantics.

The two event pairs of shared memory are often used in tandem. Ether addresses are logically organized into structures, and each structure has another address designated as a lock. A processor posts an `acquire` event for the lock of a structure before posting any `read` or `write` events for addresses within a structure, and follows reads and writes with a `relinquish` event for the lock. This protocol ensures that at a `write` event will not be posted concurrently with an other event to the same address, and allows all events between the `acquire` and `relinquish` to be considered as an atomic event. The protocol is relaxed when other aspects of the algorithm ensure that reads and writes to the same address are correctly ordered—e.g. when all further accesses to a given address are `reads`. The protected data structure has some similarity to a large-granularity event, but is different than message passing in that it is non-buffered, the lock events are not partitioned, and when the locking mechanism is not used, multiple processors can post concurrent `read` events to the structure elements.

There are some existing MIMD architectures which do not fit our ether model. Some examples are those which perform synchronous message passing, in which the processor posting the primary event also stalls until the event is matched, and architectures which have "full/empty" bits associated with their addresses that do not have fixed semantics—i.e. a primary event may set either set or clear the bit, a secondary event may wait for the bit to either be set or cleared.

Although the semantic combinations described here are, to some extent, driven by physical characteristics, and therefore by architecture, portability is rapidly becoming an important factor. Implementations of shared-memory semantics on high-latency ether [29] and message-passing semantics on low-latency ether are growing more popular. Usually, caching is used to compensate for high-latency ether.

This work will attempt to develop an ether semantics which addresses the needs of both high- and low-latency ether. To accomplish this, we keep overhead low by supporting small, fixed-size data events while providing high-level destructive event pairs for acquiring ownership of a large number of ether addresses at one time, allowing them to be transported when ownership is established in a high-latency environment. We support a more relaxed version of partitioning which allows a single address to be associated with a subset of processors, thereby allowing complete partitioning to address latency issues on those occasions when the processor posting the secondary event is known in advance, but do not require partitioning for the other occasions. We provide enough information for an implementation of the model to optionally provide buffering when it will not affect the semantics, but the semantics do not include buffering.

1.3. Parallel Algorithms

According to Knuth [28], an algorithm is a set of rules which gives a sequence of operations for solving a specific type of problem. Even if we avoid questions of how a set can "give" a sequence, and whether the rules of an algorithm truly form a set, this definition states that an algorithm is a means of expressing a sequence of operations. By this definition, a parallel algorithm is not an algorithm at all. We therefore weaken this definition somewhat, defining an algorithm as a means of expressing a set of computations for solving a specific type of problem. A computation will be the representation of a function which maps some input and initial state into some output and final state. If the set of computations produced by an algorithm always contains exactly one element, we call the algorithm deterministic, otherwise we call it non-deterministic.

sequences of operations which become the computation. The semantics of the language in which the algorithm is expressed provide the rules for unfolding the algorithm into a computation. This similarity of algorithm and computation aids in understanding their correspondence. It is our thesis that parallel algorithms can be made easier to understand if their representation more closely resembles a folded partial-ordering of operations. To allow an algorithm to be nondeterministic, the semantics of our method will allow some algorithms to be unfolded in more than one way.

Applicative (functional) and data flow languages provide one starting point for this work, in that they also express computations which are partial orderings of operations, and resemble their computations mathematically. Unfortunately, these languages have some characteristics which make them poor choices for representing parallel algorithms:

(1) The operations (primitive functions) in these languages are small (i.e. execute in a very short time). Cues for determining how these operations should be scheduled on processors for maximum efficiency are not apparent from the partial ordering alone.

(2) The languages use single-assignment (or no-assignment) variables to signify data dependences. This paradigm provides few cues as to how memory can be used and re-used efficiently.

(3) They are deterministic.

The first problem can be addressed by simply increasing the size of the operations. But as the operations get larger, the amount of data that they consume and produce also gets larger. This aggravates the second problem, since an operation updating a small part of a large data structure needs to create a new copy of the data structure as a result to preserve referential transparency.

As operations become larger, they also consume more inputs and produce more results. Although some applicative languages allow an operation (i.e. function) to return several results, the textual (linear) form of these languages does not provide for a natural representation for composing these functions.

These drawbacks to large operations will be addressed here by allowing an operation to return several results[3], and by allowing a single variable to serve as both an argument and a result to an operation. Even so, an operation will specify a function from its inputs before evaluation to its outputs upon completion. Traditional functional composition is no longer suitable for operations of this form, so an alternate form will be introduced.

The ability to use a single variable as both the input and output of an operation rules out the use of single-assignment variables to coordinate operation executions. One possible alternative method is to assign each operation application (called an *instruction*) a condition, or guard, which must be satisfied for the instruction to execute. Unfortunately, the generality of guards obscures the possible affects that the execution of one instruction can have on the ability of others to execute. The partial ordering defined by the parallel algorithm becomes obscured by the multitude of possible combinations of values of variables. Values become overloaded, being used for both the data being transformed by computations and as a direct means of controlling the execution.

Single-assignment variables and guards are similar in that they both rely on the state of some set of variables to determine whether an instruction can execute; either the "defined-undefined" state of single-assignment variables, which we will term *control state*, or the values within variables, which we will term *data state*. The excessive power of guards

[3] This is an entirely different tack than is taken with curried functions, where each function takes one argument and produces one result (which may be another function).

comes from their ability to depend on so many more possible states per variable, and to depend on variables that are not used by the instruction being guarded. The restrictiveness of single-assignment variables results from the small number of possible states (i.e. two) and from the specific semantic meanings assigned to those states which is not under the algorithm's control. A middle ground can be achieved by extending variables to have more possible control states, allowing control state to serve as a simplified form of guard.

Removing the fixed "defined-undefined" usage of control states requires that we manage the control states explicitly. An instruction in our model will specify the control state which each of its variables must have to enable execution, and will determine the final control state that those variables will be left with after execution. Non-determinism is introduced as it would be with guards, by allowing different instructions to access the same set of variables under similar conditions (identical control states). To preserve as much information as possible about the effects of each instruction on the control state of its variables, the range of possible final control states for each instruction will be made explicit.

Another cost of removing the special semantics of the "defined-undefined" control states is efficiency. When a single-assignment variable has been defined, it can be read by many functions concurrently with impunity, and the next version of the variable can be pre-calculated. We will provide techniques that allow buffering and multiple-readers in an implementation, even while the model presented to the user provides atomic access to variables.

These extensions to functional languages remove some of the mathematical resemblance of algorithms to their computations. In its place, we look toward a topological resemblance. If a parallel computation is a partial ordering and thus two-dimensional, it is reasonable to assume that a folded computation would also be two-dimensional (i.e. a graph). It is this syntax which gives our model its name: Function Networks, or F-Nets. The exact form of a computation and the semantics for unfolding an F-Net into a computation will be described. A textual syntax will also be provided, where each instruction resembles a subroutine call.

1.4. Preserving Sequential Semantics

The most important part of the specification for a stand-alone sequential algorithm is the input-output mapping that it implements. Provided that the algorithm is not real-time or interactive, factors such as the speed with which it performs that mapping, or the order in which inputs are accepted from different sources or outputs are produced to different sites, do not alter the correctness of the algorithm, though they may affect its intrinsic value in comparison with other algorithms which implement the same input-output mapping. We call a model *seperable* if the input-output mapping expressed by each sequential process is the only characteristic which affects its behavior within a concurrent model. A direct result of this property is that the implementation of that mapping has no effect on the overall semantics, so each sequential algorithm can be implemented separately, using only its partial function as a specification. Another is that all the existing technology, methods, and languages which exist for developing, expressing, and analyzing sequential algorithms can be utilized.

Separability infers other properties. Since the input-output mapping enforced by each sequential algorithm could conceivably be implemented by first reading all inputs then producing all outputs, a separable model cannot depend on concurrent execution of the sequential algorithms: There must be a serial schedule in which the algorithms could be executed. In addition, if an algorithm has multiple input sources, they simply represent multiple arguments to the function expressed by the algorithm. Since neither the order in which these arguments will be "read" nor the time that it will take the algorithm to evaluate is part of the description of the partial function, the execution of the algorithm must be considered to be atomic: it either evaluates completely, or does not evaluate at all.

Separability, and therefore serializability, has an important implication with regards to portability. If the functions expressed by the sequential algorithms are total[4], each algorithm will necessarily complete in a finite amount of time, so the parallel program can be executed without context switching, no matter how many or how few processors are available in a target architecture, increasing portability. Since it is undesirable to restrict sequential algorithms to express total functions, and nearly always impossible to show that they are, this finding is of limited utility, but if it is assumed that algorithms are in fact total in most practical cases, they can be executed without context switching by default, invoking context switching only if available processors are monopolized by long-running algorithms.

1.5. Conclusion

A description and rationale for the F-Net model of computation is presented in the remaining chapters. F-Nets are designed to function well with either high- or low-latency ether. The model is separable, so traditional sequential languages can be used for the bulk of the code. Algorithms in the model can be expressed in a graphical form resembling the folded partial orderings of their computations.

Chapter 2 describes related work which addresses many of the same goals. Chapter 3 builds the F-Net model from scratch, with design decisions driven by the above goals.

[4] They also need to be continuous, in the denotational semantics sense. This is assured by the fact that they are expressed in a language with continuous semantics.

Chapter 4 provides a formal semantics for the model, expressed mathematically as a set of axioms. This includes defining the form of a computation in the model. Chapter 5 relates the F-Net model to other formal models of concurrent computation. Chapter 6 develops efficient and correct implementations of the model to run on top of ether with shared-memory and message-passing semantics. Chapter 7 addresses some of the shortcomings of the model, proposes how the model could be extended used as the basis for new tools and architectures for parallel processing.

CHAPTER 2

Related Work

2.1. Introduction

The first chapter provided an overview of the goals of this work. This chapter will describe other models which have been developed to address some of the same goals. Unfortunately, this includes virtually all work performed in the field of parallel processing. We attempt here to sample many of the techniques and relate them to those goals, concentrating on those that have the most similar set of goals to F-Nets.

The difference between a model and a language is not clear. A language can be considered as a user-oriented model, and the role of a language processor is to convert a user-oriented model to an architecture-oriented model. Judging whether a model is architecture-independent therefore requires knowledge of the language processor, the characteristics of programs which it processes, and the similarity of the user model to an architecture model. This chapter will make no attempt to address the first two of these three factors.

The models discussed here will each be broken into their process models and their ether models. A common process model consists of sequential processes which post events to the ether. The method of initiating processes may vary, but there are commonly no restrictions on the events that a process may post nor on the persistence of the process. The processes are typically expressed in a traditional sequential language, though these languages sometimes require minor extensions to provide the capability to post events. We call such a process model the *traditional* process model.

If a parallel computation model provides a traditional process model and unrestricted access to the ether, the model will not demonstrate separability. This can be seen from the fact that deadlock is possible between processes, since each may post primary events and each may post secondary events in reverse orders. Thus, the behavior of a process depends on the implementation of its input-output mapping—i.e. the order in which it posts events.

2.2. Shared Memory and Message Passing

Most current parallel computer architectures are designed to support an ether with one of the semantic combinations described in the first chapter: shared-memory or message-passing. These architectures are usually supplied with *multitasking* tools to facilitate a traditional process model for posting events to this native ether. Message-passing events and `relinquish` and `acquire` events for shared memory are often implemented by subroutines, while `read` and `write` events for shared memory are usually implemented by utilizing a loader to map some portion of the ether addresses into the process address space, at which time ordinary memory operations on those addresses are used.

When architecture-independence is not an issue, multitasking provides low-level control over the native ether. Because this approach is so popular, several tools have been developed with the goal of standardizing the interface to these native ethers and providing some more complex ether events, such as barrier and broadcast, which can be built directly from the low-level ether events. These tools are meant to ease programming and address independence of number of processors, but are not meant to be ether-independent. For shared-memory ethers, these tools include The Force [24], the monitors package from Argonne National Laboratories [8], Large-Grain Data Flow [6], and Schedule [16]. Packages for message-passing ethers include the messages package from Argonne. A restricted form of Schedule [7] has also been implemented for message-passing ether.

An ether with message-passing semantics can be implemented relatively easily on top of an ether with shared-memory semantics by using a subroutine library to manage the ether, and to match and copy events. However, the overhead required to copy events to and from ether, the large granularity of events, and the required partitioning of the address space make this an ineffective programming environment for low-latency ether. To update a small portion of a large event in the ether, the entire event must be copied to the process, altered, then copied back to the ether. If multiple processes wish to read the same large event concurrently, two events must be posted and matched. We will refer to these drawbacks as the *update-in-place* and *multiple readers* problems, respectively.

An ether with shared-memory semantics, often called Shared Virtual Memory (or Virtual Shared Memory) [29], can be implemented on top of an ether with message-passing semantics by passing messages whenever an attempt is detected to access a shared-memory ether address which is not currently local to the posting processor. To provide sufficient granularity of events, implementations rely on passing and caching several sequential shared-memory ether addresses at one time as a *page*. The number of page transfers can be minimized by implementing a causal ordering [2] which respects shared-memory semantics but has the effect of arbitrarily postponing some processes. In some implementations, these postponements can be indefinitely long. The efficiency of these methods depends a great deal on the locality of access within the virtual shared-memory address space.

2.3. Parallelising Compilers

A parallelizing compiler takes a standard sequential program and produces a parallel program with identical input-output behavior. By definition, this approach demonstrates separability but limits expressibility: e.g. non-deterministic programs cannot be expressed. Efficient use of processors and ether is completely determined by the capabilities of the compiler. Currently, efficient mapping can only be performed for loops, and only for shared-memory ether [4].

Some compilers accept a slightly extended version of a sequential language (usually Fortran) in order to allow the programmer greater expressiveness. Adding these special constructs to a program often does not alter its semantics in any way, but informs the compiler of locations in the program where opportunities for parallelism might be found. In some cases, these extensions add small amounts of non-determinism to the program in order to increase parallelism. In most cases, these *microtasking* extensions apply only to Fortran DO loops and produce object programs for shared-memory ether [27]. Similar extensions for message-passing ether are still in the experimental stages [10].

Another method for the programmer to augment the information present in the source code is through the use of interactive parallelizing tools such as R^n [3] and Faust [21].

2.4. Linda

Linda [11] presents a traditional process model, and an ether model called *tuple space*. In this ether, events do not include addresses per se—the secondary event matches a primary event based on characteristics of the data value in the event. There are two event pairs defined for the ether, `(out, in)` and `(out, rd)`, which share the same primary event. These pairs have large granularity, are buffered, and are non-partitioned. The first pair is destructive, the second is not.

The lack of addresses in the ether model does not correspond to any common hardware implementation of ether. To compensate, the designers of the model propose methods for determining an address or partial address (i.e. hash table bin) for some events based on a global analysis of all event postings within the algorithm. The same analysis can sometimes yield a partitioning for these addresses. In cases where a complete address cannot be determined in this way, a look-up must be performed (at run-time) to match events.

Tuple space offers benefits over message passing by avoiding explicit partitioning, providing a more natural interface to high-latency ether on demand-driven applications. When partitioning is beneficial, it can sometimes be computed without user involvement. However, the associative matching of events incurs overhead in both low-latency and high-latency implementations, and the multiple-readers and update-in-place problems are not addressed for low-latency environments. The fact that the events cannot be partitioned in some cases magnifies the effect of latency in high-latency environments.

2.5. Unity

Unity [12] provides only a process model. The ether model is identical to memory (i.e. shared-memory minus the (`relinquish, acquire`) event pair). Each process consists of an optional guard and a deterministic calculation. There is no synchronization: a program execution consists of attempting to execute each process infinitely many times, in no particular order. A process execution will succeed, reading some ether addresses and writing some other (not necessarily disjoint) ether addresses, if and only if the guard is satisfied. The result of a computation is defined as a fixpoint of this computation—i.e. the state of the ether when no further changes can occur through further computation.

The execution of each process is atomic by definition, so this model demonstrates separability. The model was originally presented as a teaching tool, and presents a computation as a non-deterministic sequence of operations rather than as a partial ordering. The role of a language processor will be to determine efficient partial orderings from this specification, and to use high-latency ether effectively. There is little in the model itself to ensure that these will be possible.

2.8. Reactive Kernel

The Reactive Kernel [5] implements a traditional process model, and an ether model which consists of a memory semantics plus a (`xsend`, `xrecv`) event pair which is buffered, partitioned, and destructive, similar to a message-passing semantics. The latter event pair is used to transfer a *capability*, which can be considered as a "key" which "unlocks" some range of ether addresses. A given capability can be held by only one process at a time. There also exists an ether server which dispenses and collects capabilities.

A process cannot post `read` or `write` events for an ether address unless it holds a capability which includes that address. The intent is that when a process wishes to pass some state to another process, it acquires a capability from the server, posts data events, then passes the capability to another process by posting an `xsend` event containing the capability. When a process wishes to acquire state from another process, it acquires a capability by posting a `xrecv` event, then performs data events before passing the capability back to the server or on to another process.

In a high-latency environment, all of the data events described by a capability can be transported through the ether with the capability. In a low-latency ether, all data events can be left stationary. The update-in-place problem is addressed by this model, but the multiple-readers problem is not.

2.7. Specification Languages

Petri nets [36] are a prime example of a model which has partial orders as computations and in which "algorithms" are the folding of those computations. Simple Petri nets do not provide any notion of data, and therefore do not present an ether model, but the do provide a natural representation of concurrency. They are used primarily as a specification language for concurrent processes.

Some data extensions to Petri nets which have retained many of the Petri Net semantics are Macro E-nets [34], which support high-level Petri-net-like constructs designed to model computer architectures, and VLP [19], which is presented as a method for introducing synchronization into specification-level dataflow diagrams.

CODE [39] is a high-level specification language for parallel processing, allowing the user to specify dependences and exclusion between abstract processes.

2.8. Dataflow Languages

Dataflow is a general term used to refer to networks of processes connected by buffered streams. In terms of the ether model we have discussed, the semantics of a stream ether consists of a `(define, use)` event pair which is buffered and destructive. Unlike that ether model, both the define and use events are partitioned: i.e. a stream can be written by only one process and read by only one process. This provides for demand-driven semantics, and always results in deterministic programs (providing the composite processes are deterministic) [25]. A primary selling point of dataflow languages (e.g. SISAL [30] and Id) is the fact that they demonstrate separability.

The processes in a dataflow model traditionally consist of elementary arithmetic operations. Dataflow languages provide the ability to compose functions using standard functional composition and single-assignment variables. The (define, use) event pair is not efficiently implementable in high-latency ether because of its fine granularity. In low-latency ether, the destructiveness of the event pair causes undue copying. In either case, the fine granularity of the processes causes excessive overhead due to scheduling. Special architectures [35] are being constructed to minimize process scheduling overhead and overlap computation with communication latency. More practical approaches for low-latency ether involve compiler technology to increase the granularity of operations and re-use memory efficiently [20].

2.9. Coarse Grain Data Flow

The efficiency and use of ether can also be increased by allowing the user to create larger processes explicitly, often in a traditional imperative language, thereby increasing the granularity of ether events. Examples include MUPPET [33], Loral DGL [26], and TDFL [40]. These models retain the problems intrinsic to message-passing on both high- and low-latency ethers. Non-determinism is introduced in some of these by allowing some operators to execute when only some subset of their arguments are present, thereby allowing streams to be arbitrarily merged.

2.10. Actors

Actors [1] is a theoretical model of processes (actors) which correspond via messages in a very restricted framework. In this model, each actor receives messages through a single message queue. As a result of reading a message, an actor can create other actors, send messages to other actors, and define a new behavior for itself (i.e. to process its next message). The resultant actor program is constantly evolving, with new actors being created and old actors changing behavior. A primary strength of the actor model is in its formal treatment of message-passing.

2.11. Strand

Strand [18] programs resemble logic programs, consisting of a set of clauses, each consisting of a head, a guard, and a body. All data is passed using single-assignment variables.

During execution, process calls are deposited to a "process ether", with parameters consisting of addresses in the data ether. If a call matches the head of a clause, the variables of the clause head are bound to the same ether addresses as the arguments of the process call, all other variables in the clause are bound to new data ether addresses, and the guard of the process is checked. If the guard is satisfied (which requires that all variables in the guard have been defined), the process call is removed from the process ether, and the process calls declared in the process body are deposited. This continues until primitive processes are called. A primitive process has well-defined in and out arguments, and executes atomically to define the out arguments based on the in arguments. Primitive processes are supplied as part of the model, though some implementations allow the user to introduce tailored primitive processes implemented in a sequential language.

All three steps—head matching, clause checking, and reduction—are defined to occur as a single atomic action. Thus, Strand demonstrates separability. Strand is essentially a dataflow language in which functions can have multiple results as well as non-determinism (since the heads and guards of many process definitions may allow many different reductions to occur). Because of the single-assignment paradigm, this model retains the update-in-place problem of traditional dataflow.

2.12. Paralation

Paralation [38], like Linda, relies on copying and associative lookup for its architecture-independence. The model actually consists only of a few general data-reduction operators, which are targeted for implementing algorithms based on data parallelism (SPMD and SIMD), but are specifically not designed to aid communication between non-homogeneous processes, which is central to MIMD computing.

2.13. Synchronous Models

Synchronous models such as Occam [23], CSP [22], and CCS [32], provide an ether with message-passing semantics, but one in which a primary event will stall if there is no matching secondary event in the ether. Thus, the ether is never required to store unmatched events: it serves only as a means of communicating from the poster of the primary event to the poster of the secondary event. The increased likelihood of a stall in this ether is addressed in CSP and Occam by accompanying it with a fine-grained process model which allows the programmer to offer alternate work whenever a process stalls.

Synchronous communication is not a real-world phenomenon, and must be built from asynchronous communication. If processors consume space and each processor can post at most one event at a time, then the posting of events by different processes must occur in a different space or at a different time. Thus, for the sender to obtain the message and for the receiver to be informed that it has been received requires two communications.

Any process which contains a send or receive can stall indefinitely depending on the behavior of other processes. This could be considered as violating separability. It is the role of CCS to define the behavior of each process in its environment.

2.14. Historical Perspective

The next section will present the F-Net model as though its features were derived directly from the goals set forth in chapter 1. In fact, the model was derived as an attempt to formalize the semantics of Babb's LGDF technique, modifying or removing those portions of the model which made its implementation difficult using shared-memory or message-passing semantics. This led to the LGDF2 [15] technique and the F-Net model. The LGDF work, in turn, was developed to provide execution semantics for Data Flow Diagrams such as those used in Structured Analysis techniques [13], for the dual purposes of providing a smooth transition from design to implementation and of developing a parallel programming method for shared-memory MIMD computers. This work was also influenced by the work of Browne. The SCHEDULE model has similar lineage. The VLP model, which is similar to F-Nets, was developed independently, also with the goal of providing semantics to high-level data flow diagrams.

CHAPTER 3

F-Nets

3.1. Introduction

The first chapter presented some of the goals of this thesis: to define a model for parallel computation which is architecture independent, provides separability, and expresses an algorithm as a folded partial ordering of operations. This chapter will build the model, called F-Nets, providing some justification for each step in terms of those goals.

The chapter consists of two major sections. In the first, the basic constructs of the model are developed by addressing the goals listed above. In the second, methods of including more high-level information in the F-Net are examined, both in terms of how this information provides the programmer with a more abstract view of the F-Net's behavior, and in terms of how a scheduler can take advantage of the information to optimize performance.

To give some grounding to the points made in this chapter, a sample problem will be presented—to compute

$$\sqrt{\sum_{i=1}^{50} i^2}$$

The computation will proceed by initializing a shared variable, i, to 51 and a shared running sum to 0, then having two worker processes repeatedly decrement i, square the new value, and add the result to the running sum. When all is finished, another process will take the square root of sum to produce the final answer.

3.2. Building the Model

In this section, we construct the major components of the F-Net model.

3.2.1. Architecture-Independent Ether Model

We begin by defining an ether model which addresses two of the three latency-related semantic differences presented in the first chapter: granularity and partitioning. Buffering will be addressed later.

High-latency ether benefits from large-granularity events so that more data is transferred each time the latency of the ether is experienced, while low-latency ether benefits from fine-granularity events to keep overhead low. We can accommodate both of these requirements by presenting the ether at two levels of granularity. The fine-grained data addresses are grouped into collections which we will call *m-variables* (or simply variables, when no confusion will result). For the time being, assume that each m-variable has an associated address designated as a lock. There are two event pairs defined for this ether: a non-data event pair, `(orphan, adopt)`, which is unbuffered, non-partitioned, and destructive and operates on a lock address, and a fine-granularity event pair, `(write, read)`, which is unbuffered, non-partitioned, and non-destructive and operates on a data address. A process cannot post any data events for an m-variable address until after it has posted an adopt event for the lock.[1] When a process is finished posting data events, it may release the m-variable by posting an orphan event for the lock.

This very closely models the way that shared-memory semantics are often utilized. In this low-latency environment, the m-variable as a whole can remain stationary in the ether, and all events can be implemented with fine granularity. In a high-latency environment,

[1] Whether this protocol is enforced by the ether itself or by the processes independently is unimportant, as long as it is enforced.

the adopt event can be used to move the data (i.e. pending data events) associated with the m-variable as a whole through the ether, with fine-grain data events occurring local to the processor which successfully adopts the variable.

Partitioning can be added by extending the lock associated with an m-variable to a set of locks, only one of which can be orphaned at any one time, and partitioning the (orphan, adopt) event pair. Thus, when an orphan event is posted, the process which can perform the adopt event will be known.

For the F-Net model, we consider partitioning too restrictive for reasons mentioned earlier. Instead of mapping each lock address to a unique process, we map each to a set of processes. In those cases where this set contains but one element, this scheme is identical to conventional partitioning. In other cases, some transport of the m-variable may still be possible in a high-latency environment to a point closer to all possible adopters. In either case, the scheme has software-engineering benefits over a non-partitioned scheme, since inappropriate processes are not able to adopt the variable.

We also require that sets of processes to which the locks of an m-variable are mapped must be disjoint—i.e. that each process has at most one lock of an m-variable mapped to it. This simplifies the protocol required for a process to adopt a particular m-variable, thus reducing overhead. When the model has been completely described, it can be seen that this restriction does not restrict expressiveness.

The data associated with an m-variable (i.e. the pending fine-granularity data events) will be called its *data state*. The set of lock addresses associated with an m-variable will be called its *control domain*, and the element of that domain which was last orphaned, if any, will be called the *control state* of the variable.

An m-variable can be considered less formally as a bin which can contain a data structure (its data state) and which carries a flag which lists the processes which can adopt it (its control state). An adoption request for the variable will block until the control state of that variable contains the name of the adopting process. On successful adoption, the control state of the variable will be atomically cleared, which will keep any other process from adopting it, and the data state will be made accessible to the process. When finished accessing the data state, the process may relinquish access to the variable by orphaning it, at which time the process must also specify a new control state.

To demonstrate the use of m-variables, we give a first approximation of a solution to the sample problem. i and sum must be represented within m-variables, since they are communicated among several processes. Although both could be kept in the same m-variable, contention can be decreased by putting them in separate m-variables, since they will be needed by both workers in different phases of their execution.

We will write the processes in a C-like sequential pseudo-code, augmented with two special statements:

```
adopt x
```

which will post an adoption request for the appropriate lock of m-variable x, and

```
orphan x as y
```

which will orphan m-variable x with a new control state of y. When an m-variable is otherwise mentioned within the pseudo-code, it will play the role of a variable which refers to the data state of the m-variable. We name the m-variables `i` and `sum` to reflect the data stored there. The final answer will be deposited into an m-variable named `ans`.

The processes will be named `init`, `worker1`, `worker2`, and `finish`. We use a fictional process, `therest`, to symbolize the destination of the answer after `finish` produces it, so the end of the program will be signified when the m-variable ans obtains a control state corresponding to this process.

An attempt at the problem solution follows. This is not a legal F-Net, but illustrates the use of m-variables.

```
Variables
  int i        ( uninit = {init}, valid = {worker1, worker2} );
  int sum      ( uninit = {init}, valid = {worker1, worker2} );
  float ans    ( empty = {finish}, full = {therest} );

Processes
  init
  {
    adopt i;
    i = 51;
    orphan i as valid;
    adopt sum;
    sum = 0;
    orphan sum as valid;
  }

  worker1
  {
    int temp;

    adopt i;
    i = i - 1;
    temp = i;
    orphan i as valid;
    temp = temp * temp;
    adopt sum;
    sum = sum + temp;
    orphan sum as valid;
  }

  worker2
    (Same as worker1)

  finish
  {
    adopt sum;
    adopt ans;
    ans = sqrt ( sum );
    orphan sum as valid;
    orphan ans as valid;
  }
```

The m-variable declarations at the beginning list the type of the data state and the control domain for each m-variable. The processes which correspond to each element of the control domain are also listed. The first declared element of the control domain is the initial control state.

The problem with this code is that the workers do not stop working when i reaches 1, and the `finish` routine cannot ever adopt `sum`. It is tempting to change the last line in the workers to

```
if temp == 1 then
   orphan sum as done
else
   orphan sum as valid
```

where `done` corresponds to the `finish` process, but the order in which the workers adopt `i` is not necessarily the same as the order in which they adopt `sum`. The solution to this problem will be dealt with later.

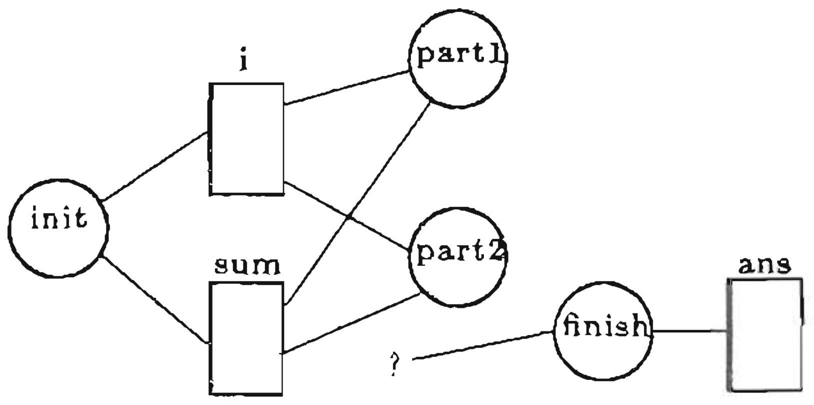


Figure 3.1 First Attempt at the Sample Problem

An F-Net is illustrated graphically by showing each process as a circle, and each m-variable as a polygon, as shown in Figure 3.1. The sides of an m-variable polygon represent the elements of the control domain. (Variables with a control domain of only one element are shown as a line, those of only two as a rectangle.) The side representing the initial control state of the m-variable is identified by using a thicker line. An arc connects each process to the m-variables which it can possibly adopt: The polygon side to which it is connected corresponds to the element of the control domain containing that process. This notation makes the control domain of each variable apparent from the diagram.

Note that for maximum parallelism, the workers must maintain access to their m-variables for as little time as possible. To facilitate this in the example, a copy of `i` is kept in `temp`. If `i` contained a large data structure to be accessed, it might have been more favorable to retain access to the m-variable, decreasing copying time at the expense of decreasing parallelism.[2]

3.2.2. Separability

We attain the goal of separability by breaking a process into segments, each of which has the same semantics and halting behavior as it would if it were executing alone. In other terms, we wish the computations described by the segments to be atomic transactions—i.e. computations that can be serialized (since the executions are independent) and either never begin or complete (since their halting behavior is independent of other transactions). A set of transactions is assured to be serializable if and only if each transaction is two-phase [17]. Two-phase, in the case of F-Nets, means that each transaction can be divided into a growing phase containing no orphan events, followed by a shrinking phase where all m-variables

[2] Other approaches which do not incur copying or decreased parallelism will also be made possible when buffering is introduced.

which were adopted in the growing phase are orphaned.

The bijection above requires that segment executions have this form if they are to have the desired properties. A segment which posts an event stream which is not of this form can always be modified to be of this form: If additional m-variables are needed after orphaning some but not all of those that it owns, the transaction can terminate by orphaning all of the rest of its m-variables and initiating a new segment which begins by re-adopting these variables as well as the newly-needed variables.

By definition, an atomic transaction must complete if it shows any evidence of beginning execution, but we do not wish to restrict our segments to express terminating computations. We can consider non-terminating computations as atomic transactions by asserting that each m-variable has an additional element of its control domain called $\perp$ which does not correspond to any process. In terms of the interactions of other processes with the ether, there is no difference between a segment which orphans an m-variable with a control state of $\perp$ and one which does not orphan the m-variable at all: In either case, no process can adopt the m-variable. A non-terminating segment (i.e. one which does not orphan some of its m-variables) can therefore be modeled as one which orphans some of its variables with a control state of $\perp$ (pronounced "bottom"). The $\perp$ control state will be represented in our system as the absence of any other control state.

Atomicity guarantees that a transaction will not partially complete, but it does not guarantee that it will ever begin, even if there is nothing to stop it from doing so. If the model is to provide the power to demonstrate that programs will execute and produce results (liveness) as opposed to simply limiting the possible results (safety), the conditions under which transactions will execute must be made explicit. We do so here:

> A transaction will not be indefinitely postponed if the conditions required for its execution (i.e. the proper control state of the m-variables which it adopts) will be met continuously until it executes.

This rule has the effect of ensuring that a transaction will not be postponed indefinitely due to deadlock.

Liveness (and specifically absence of deadlock) is related to separability, though it is not usually regarded as such. If the issue of deadlock is avoided at the level of the model, to be addressed at the algorithm level, the correctness of any new segment (in the sense that it preserves liveness of the program as a whole) could depend upon the order in which variables are adopted with respect to other segments. Specifying liveness at the level of the model itself avoids this problem.

The model will not strand the implementor with the very difficult problem of deadlock avoidance. We will require that each segment adopt all of its m-variables in one atomic action. In this way, an implementation can ensure that all m-variables have the correct control state before a segment adopts any of them.

3.2.3. Independence of Number of Processors

The liveness rule in the previous section ensures that segments which can execute will eventually be executed, regardless of the number of physical processors in the architecture, or the number of transactions to execute on those processors. The ability to adopt all m-variables in one atomic action aids an implementation in efficiently sharing physical processors among processes, since each segment can be made to block exactly once, then execute to completion, minimizing the need for context switching.

The overhead required by a context switch can be reduced even further by forbidding the passage of any persistent local state (i.e. program store, registers, and program counter) from one transaction in a process to the next. Explicit passage of persistent local state between transactions can be handled the same way as the passage of any other data state; via an m-variable. The data state of an m-variable is persistent by definition, and 'local' means only that the control domain of the m-variable contains only one control state, which is mapped to a single process.

The program counter can also be mapped to the control state of an m-variable. To illustrate this, consider the code of a process which has *n* statements which mark the beginnings of segments—i.e. where all of the m-variables for the transaction are atomically adopted. If *n* different processes are constructed, each identical to the first except for an initial branch to one of the segment beginnings and the additional adoption of a common m-variable (say, STATE), each transaction can select the next process (and therefore segment) to execute by orphaning STATE with that next process as the new control state. The control state of STATE is now effectively the persistent program counter for the original process between transactions.

When persistent local state is ruled out, a process per se has no role. All state and control is managed explicitly by the segments. If more state is carried between segments of different processes than is carried between segments of the same process, the concept of process becomes somewhat confusing. The F-Net model will therefore not include the concept of a process: an algorithm will consist of a set of these segments, called *instructions*. Since an instruction is deterministic, has no state before adopting its variables, and adopts all of its variables in one atomic action, it must adopt the same set of variables on each execution. By making this set of variables part of the specification of the instruction, the adopt construct can (and will) be omitted from the model.

3.2.4. Sample Problem

In the sample problem, each worker consists of two transactions. It would be possible to unite those two into one by adopting both m-variables first, but this would counteract the parallelism achieved by having two separate m-variables. Since the init process is only executed once, it will simplify its execution to merge its two transactions. We present the sample problem again below and in Figure 3.2. This time, the code is correct, and is structured just as it will be in the final model.

```
Variables

    int   i      ( uninit inrange outrange );
    int   hold1  ( empty full );
    int   hold2  ( empty full );
    int   sum    ( uninit valid );
    float ans    ( empty full );

Instructions

  init [ uninit i, uninit sum ]:
    i = 51;
    orphan i as inrange;
    sum = 0;
    orphan sum to valid;

  part1 [ inrange i, empty hold1 ]:
    i = i - 1;
    temp = i;
    if (i > 1)
      orphan i to inrange;
    else
      orphan i to outrange;
    hold1 = temp * temp;
    orphan hold1 as full;

  part2 [ inrange i, empty hold2 ]:
    ...
    hold2 = temp * temp;
    orphan hold2 as full;

  red1  [ full hold1, valid sum ]:
    sum = sum + hold1;
    orphan sum as valid;
    orphan hold1 as empty;
```

```
red2  [ full hold2, valid sum ]:
  sum = sum + hold2;
  orphan sum as valid;
  orphan hold2 as empty;

finish [outrange i, empty hold1, empty hold2, valid sum, empty an

  orphan i as outrange;
  orphan hold1 as empty;
  orphan hold2 as empty;
  ans = sqrt ((float) sum);
  orphan ans as full;
  orphan sum as valid;
```

Note that the two transactions in each worker have become two separate instructions, called `part` and `red` (for reduce). Since data must be carried between these instructions, additional m-variables `hold1` and `hold2` have been introduced. Also note that each instruction has been given a heading declaring the variables it will adopt and the control state of each variable which correspond to the instruction. As a result, the variable

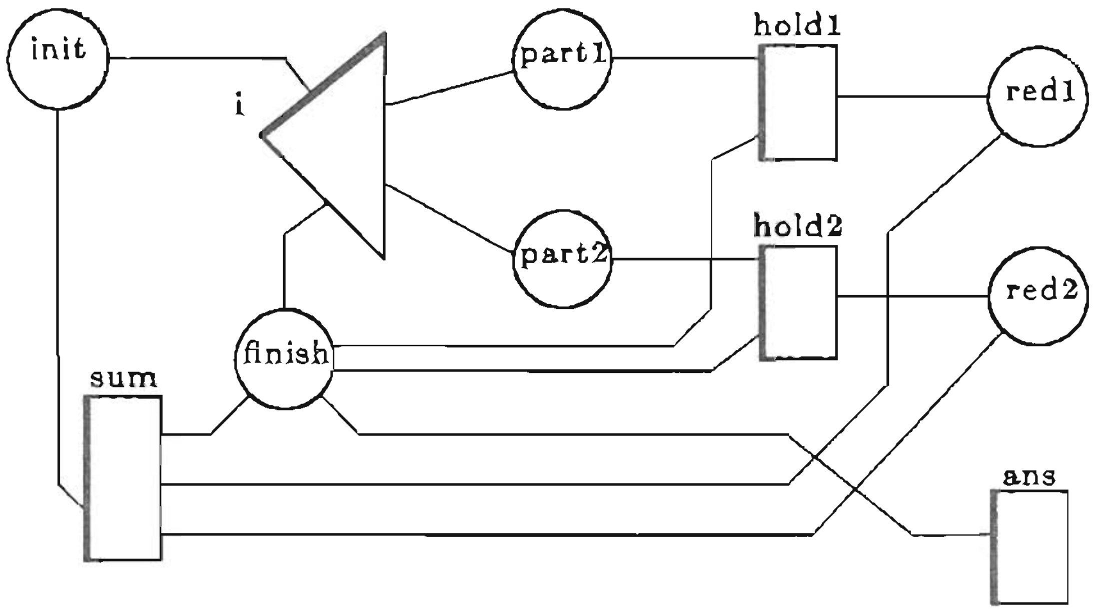


Figure 3.2 Second Attempt at Sample Problem

declaration lists only the control states of each variable, and adopt statements have been omitted from the languages.

There is now a very precise way of specifying when the finish instruction should execute: when a part process has taken the last value from i, and when no more temporary values (from the hold m-variables) are waiting to be added into the sum m-variable. Note that the finish instruction does not need to access the data states of i, hold1, or hold2, but needs to adopt them anyway as a condition that it can execute.

The final control states left by the F-Net are similar to the initial control states so that only minor modifications would be required to make the F-Net restartable.

3.3. Higher Level Characterization

In this section, we look at ways of separating high-level information about the behavior of each instruction from its implementation. This information will provide an abstract view of an F-Net that can be utilized for documentation and automated error-checking (if the low-level code is already written) or specification (if the low-level code is not). A scheduler can also use the abstract view to optimize execution.

3.3.1. Instruction Specifications

Since an instruction is deterministic and has no persistent local state before it begins executing, its behavior during execution must depend only on the data states of its m-variables when they were adopted. This behavior consists of reading and writing the data state of the m-variables, and possibly orphaning some of them with new control states. (Even if it does not orphan some of the m-variables, it will be as though it has orphaned them with a $\perp$ control state.)

Thus, an instruction can be fully characterized by the set of m-variables that it will adopt, and a function (called the *firing function*) describing the new data and control states of these m-variables based on their data states when they were adopted. This specification is complete: it describes precisely the effect of the instruction on the F-Net's state. Put another way, if instructions in an F-Net execute according to the rules already given, then any instruction implementation which fulfills this specification will exhibit exactly the same possible effects as any other.

Although such a specification is complete, some important information about how each m-variable will be used is left hidden within the firing function. Specifically,

Read usage:

Is the result of the firing function ever dependent on the data state of the m-variable?

Write usage:

Does the firing function ever specify a different new data state for the m-variable than that with which it began? (i.e. does the data state ever change as a result of executing the instruction?)

Possible new control states:

Which new control states might the firing function assign to the m-variable?

We will require that this information be specified separately, and will be called the instruction's *signature* for its arguments. It is shown in the graphical representation by introducing arrowheads on the arcs (toward the instruction for read usage, away from the instruction for write usage) and "fingers" (short arcs) within the m-variables at the end of the arcs to point to the possible new control states. See Figure 3.3. By including this information as part of the F-Net, much of the overall data and control flow can be determined without examining the implementation or firing function associated with each instruction.

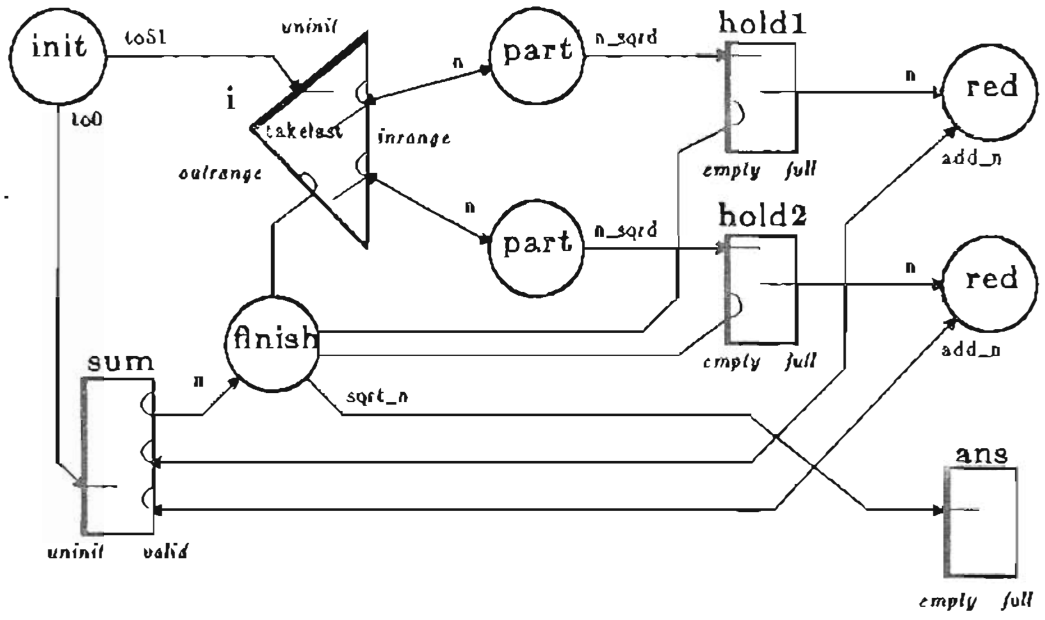


Figure 3.3 Final Diagram of Sample Problem

3.3.2. Ensuring That Signatures are Correct

The signature will be used as more than just a way of describing what an instruction does—it will be taken as the programmer's specification of restrictions on what the instruction should (and should not) do. This allows some programming errors within the implementation to be caught (e.g. the alteration of the data state of a m-variable without write usage), but also informs the run-time system, in some cases, as to how the implementation should be interpreted. For example, if an instruction implementation does not assign values to (part of) a m-variable's data state, this could be interpreted as either leaving the old values or as reinitializing the data state to some "empty" value. If the m-variable is identified in the signature as having only write usage (no read usage), then the latter interpretation must be taken, since the new contents of the m-variable cannot depend on its previous contents.

There is one case, however, where it cannot be determined whether an implementation matches its signature. If a signature declares that the $\perp$ control state is not a possible new control state for a given m-variable, this is equivalent to declaring that the part of the implementation which orphans the m-variable will be executed. Determining whether this is true would require careful analysis of the user's code at best, and is intractable at worst. To circumvent these problems, the $\perp$ control state will *always* be assumed to be a possible new control state for every m-variable for every instruction—with one exception, described in the next section.

3.3.3. Multiple Readers and Buffering in Ether

The semantics of the model ensure that only one instruction can adopt an m-variable at any one time. Even so, if the instruction which currently owns the m-variable will not write it (as per its signature), and another instruction which will not write it becomes ready except for the fact that the m-variable has not been orphaned by the first instruction, a scheduler could optimistically begin the second instruction, and both could read the m-variable concurrently. But if the first instruction never orphans the m-variable with the proper control state, the optimistic transaction would have to be backed out—an action which we would prefer to avoid.[3]

But if the first instruction can *promise* that it will eventually orphan the m-variable with the proper control state, the second (concurrent) instruction will no longer be optimistic. That is, even though the model logically allows only one instruction to adopt a m-variable at a time, in this circumstance an implementation could allow multiple instructions to read the m-variable concurrently with no risk of lost work.

[3] "Backing out" means to nullify any effects that a transaction has had on the state of the system, essentially "undoing" it.

This is exactly the approach we will take. If the signature of an instruction states that a m-variable will not be written, and if exactly one new control state for that m-variable is listed, the m-variable will be called *non-volatile*, and the signature will be taken as a promise that the instruction will eventually orphan the m-variable with that control state. This is the condition alluded to at the end of the last subsection where a new control state of $\perp$ is not assumed as part of the signature. If a scheduler suspects that an instruction will attempt to break its promise by never orphaning the m-variable, the scheduler may give the instruction a copy of the data state on the m-variable and assign the new Control state to the m-variable itself.

This same solution also provides for buffering, even without altering the semantics of the ether to include buffering. Suppose the contents of some m-variable (e.g. `hold1` in the sample problem) is written by one instruction (e.g. `part1`), and read by another (e.g. `red1`) which has non-volatile access to the m-variable and always passes the m-variable back to the writer. As soon as `red1` begins execution the first time, the scheduler knows that it will eventually finish, so execution of `part1` can be initiated again, even before `red1` has finished, provided that the new data state which it writes to the m-variable is buffered long enough to avoid conflicting with the version being concurrently read. Thus, some amount of buffering can be implemented by a scheduler in this case, even though buffering is not explicitly present in the model.

3.3.4. Instructions Performing the Same Operation

Different instructions may be alike in the transformations they perform to the data (i.e. their *operation*), but differ only in the m-variables and the control states of those m-variables to which they refer (i.e. their *binding*). Cases in point are the `part1` and `part2` instructions or the `red1` and `red2` instructions in the sample program. Entering and maintaining both instruction implementations seperately, in spite of the fact that they consist of almost identical code, is impractical. From another standpoint, the information that they are identical in some sense is missing from the F-Net, and could be used to advantage, both in reasoning about the F-Net and in implementing an efficient scheduler for the F-Net.

To facilitate reusing a single operation for multiple instructions, we decompose an instruction specification into its operation and its binding as follows:

Operation

An operation is an instruction (together with its signature) which has had its m-variable references replaced by formal parameters called *arguments*, and its control state references (in orphan statements) replaced by formal parameters called *transitions*. (Each transition will in some sense belong to one of the arguments.)

Binding

A binding is a means of creating an instruction from an operation by providing an actual-to-formal mapping. For each argument, the binding consists of three parts: (1) an *argument binding*, which specifies the actual m-variable to be used for the argument; (2) a *transition binding*, which specifies the actual elements of the control domain of that m-variable to use for the transitions of the argument; and (3) a *firing constraint*, which names the element of the control domain which corresponds to this instruction.

3.3.5. Final Version of Sample Problem

Now for the final version of the sample problem. It does not differ significantly from the previous version. The graphical version was presented in Figure 3.3.

```
Variables
  int   i       ( uninit inrange outrange );
  int   holdi   ( empty full );
  int   hold2  ( empty full );
  int   sum    ( uninit valid );
  float ans    ( empty full );

Ops
  init [ out   int to0        ( init )
         out   int to51      ( init )]

      to51 = 51;
      <init to51>;
      to0 = 0;
      <init to0>;
      endop

  part [ inout int n         ( takelast take1 )
         out   int n_sqd     ( put )               ]

      int temp;

      n = n - 1;
      temp = n;
      if (n == 1)
        <takelast n>;
      else
        <take1 n>;
      n_sqd = temp * temp;
      <put n_sqd>;
      endop

  red  [ in    int n         ( take )
         inout int add_n     ( inc ) ]

      add_n = add_n + n;
      <take n>;
      <inc add_n>;
      endop

  fin  [ nodata      snsr1     ( sense )
         nodata      snsr2     ( sense )
         nodata      snsr3     ( sense )
         in    int   n         ( take )
         out   float sqrt_n    ( put ) ]

      float temp;

      <sense snsr1>;
      <sense snsr2>;
      <sense snsr3>;
      temp = n;
```

```
            <take n>;
            sqrt_n = sqrt( temp );
            <put sqrt_n>;
            endop

    Instrs
      init [ toSl    : uninit   i      ( init   : inrange )
             to0     : uninit   sum    ( init   : valid )      ]
      part [ n       : inrange  i      ( takelast : outrange,
                                         takel    : inrange)
             n_sqd   : empty    hold1  ( put    : full )       ]
      red  [ n       : full     hold1  ( take   : empty )
             add_n   : valid    sum    ( inc    : valid )      ]
      part [ n       : inrange  i      ( takelast : outrange,
                                         takel    : inrange)
             n_sqd   : empty    hold2  ( put    : full )       ]
      red  [ n       : full     hold2  ( take   : empty )
             add_n   : valid    sum    ( inc    : valid  )     ]
      fin  [ snsr1   : outrange i      ( sense  : outrange )
             snsr2   : empty    hold1  ( sense  : empty )
             snsr3   : empty    hold2  ( sense  : empty )
             n       : valid    sum    ( take   : valid )
             sqrt_n  : empty    ans    ( put    : full )       ]
```

The F-Net is now presented in three parts: M-variable declarations, operation declarations, and instruction declarations.

The operations resemble the instructions in the previous example. Each is now preceded by its signature for each argument, which lists the data usage and type, the argument name, and the list of transition names. (Usages `in` and `inout` signify read usage, `out` and `inout` signify write usage, `nodata` signifies neither read nor write usage.) The syntax of the

`orphan` *x* `as` *y*

has been changed to

<*y* *x*>

where *y* is now a transition and *x* is now an argument. Note that the arguments are nouns, the transitions verbs.

The instructions now list the operation followed by the argument bindings (in brackets). The binding for each argument consists of the name of the argument, the m-variable and control state to which it is bound (i.e. the argument binding and firing constraint), and the transition bindings in parentheses.

It should be noted that although the sample problem adequately demonstrates the concepts, its small size does not provide a good justification of their utility.

3.4. Final Notes

The restriction that the elements of the control domain of an m-variable must be independent is not a restriction because of the functional nature of operations. Instead of using two different elements of a control domain as a firing constraint for an instruction, two different instructions representing the same operation can be used, each having all of the same bindings except that each uses only one of the elements of the control domain as its firing constraint. Since the fact that the two instructions exhibit the same behavior is maintained within the model (by virtue of their having the same operation), no information is lost in this reconstruction. As mentioned earlier, this restriction reduces overhead by simplifying the locking protocol for each instruction required to avoid deadlock.

In addition to the static representation of an F-Net shown in Figure 3.3, dynamic information relating to execution semantics can also be shown. By highlighting the side of the m-variable corresponding to the m-variable's control state (with no side highlighted if the control state is $\perp$), the execution rules can be stated as follows: an instruction can fire (i.e. execute) when it is connected only to highlighted m-variable sides. Execution of an instruction consists of evaluating the firing function corresponding to its operation, using the data states corresponding to its "read" m-variables (i.e. those with arrows toward the instruction circle) as arguments.

CHAPTER 4

Axiomatic Semantics and Formal Results

4.1. Introduction

Previous chapters have attempted to give a "feel" for the form (i.e. syntax) and behavior (i.e. semantics) of an F-Net, but formal reasoning requires more. This chapter begins by restating this syntax and semantics using the mathematical language of sets and functions.

Earlier, we defined a non-deterministic algorithm as one which describes a set of computations, and a computation as a functional mapping from input to output. We also stated that a reasonable representation for a parallel computation is a partial order of operations. In this section, we describe the execution of an F-Net as an Execution Graph. The set of Execution Graphs achievable by any particular F-Net is defined by a set of constraints in the form of axioms. These axioms are followed by theorems which show that Execution Graphs are indeed partial orderings and computations.

The constructs within an F-Net which introduce non-determinacy are then identified. With this in mind, a choice log is defined which captures the non-deterministic choices made during a computation. We then prove that such a choice log, together with the F-Net and its input, completely characterizes a computation. This is valuable information, since a choice log can be created during an execution with very little space or time overhead, and can be used to re-execute an F-Net, perhaps within a debugger, with the same results.

The notation used early in this chapter is shown in Table 4.1. The final entry may require further explanation. As a tuple is defined, its elements (fields) are named. Later reference to an element of a tuple may require identifying its parent. Since the use of subscripting for this purpose becomes confusing when tuples are heavily nested, as they are here, the alternate notation shown is used. It is intended to be reminiscent of record selection notation in the C computer language.

4.2. Syntax

Before presenting the formal definition of an F-Net in purely mathematical terms, we will first provide an outline in English. An F-Net of order p with alphabet Σ is a set of variables V, operations O, and instructions (which use those operations) I.

- A variable is a repository for *data state*, a data value being passed from one computation (i.e. instruction execution) to the next, and *control state*, an indicator of the set of instructions which can next access the variable. The data state of a variable will be drawn from its *data domain* and the control state will be drawn from its *control domain*. To avoid the complexities of type mismatches within the abstract model, all

Notation	Meaning
$A, B, \cdots Z, \Sigma$	Sets of various kinds, often of tuples
$a, b, \cdots, z$	Scalars or tuples
$\alpha, \beta, \gamma, \delta, \cdots, \zeta$	Functions
$\mathbf{N}$	The set of natural numbers
$\mathbf{P}(L)$	The Power set of L
$\mathbf{Z}$	The set of integers
$\|n, m\|$	$\{i \in \mathbf{Z} \mid n < i < m\}$
$S_\perp$	The set S augmented with a bottom element $\perp$ to create a flat domain.
$x \cdot w$	Field w of tuple x

Table 4.1. Notation

variables will have the same data domain, Σ, and the same control domain, $[1,p]_{\perp}$, where $\perp$ can be taken to represents the absence of control state.

- Operations denote atomic, deterministic computations. An operation possesses *arguments* which formally represent variables and each argument possesses *transitions* which formally represent some members of that variable's control domain. An argument is classified as being a read (written) argument, or having read (write) usage, if its data state[1] is ever used in (produced as a result of) the execution of the operation. A single argument can be read, written, both, or neither.

  The computation performed by an operation is described by a *firing function*, ϕ, which functionally maps the data state of its read arguments to new data state for its write arguments and transitions for all of its arguments. For any argument with write usage or multiple transitions, one possible transition for that argument is $\perp$ which can be interpreted as the absence of a transition.

- Instructions are instantiations of operations. In addition to specifying the operation to be instantiated, the instruction contains an *argument binding*, β, which associates each argument of the operation with variables of the F-Net, and a *transition binding*, δ, which associates the transitions for each argument and the control domain of the corresponding variable. An additional *firing constraint*, γ, denotes the control state which each argument must have in order for the instruction to *fire* (i.e. execute, performing the mapping specified by its operation).

An *F-Net* of order $p \in \mathbb{N}$ is a 4-tuple $f = \langle \Sigma, V, O, I \rangle$ where

Σ is the *Data Alphabet*

[1]When we refer to the data state or control state of an argument, we are actually referring to that of the variable which the argument represents.

V is the set of *Variables*

O is the set of *Operations*, $o \in O = \langle a, R, W, \tau, \phi \rangle$ where

$a \in \mathbb{N}$ is the *Arity* (i.e. # of arguments)

$R \subseteq [1, a]$ is the set of *Read Arguments*

$W \subseteq [1, a]$ is the set of *Written Arguments*

$\tau: [1, a] \rightarrow [1, p]$ is the *Transition Signature* (i.e. # of transitions/arg)

ϕ is the *Firing Function*

$$\phi: \Sigma^{|R|} \rightarrow \Sigma^{|W|} \times T_1 \times \cdots \times T_a$$

$$\text{where } T_k \equiv \begin{cases} \{1\} & \text{if } \tau(k)=1 \text{ and } k \notin W \\ [1, \tau(k)]_\perp & \textit{otherwise} \end{cases}$$

I is the set of *Instructions*, $i \in I = \langle o, \beta, \gamma, \delta \rangle$ where

$o \in O$ is the *Opcode*[2]

$\beta: [1, o \cdot a] \underset{1-1}{\longrightarrow} V$ is the *Argument Binding*

$\gamma: [1, o \cdot a] \rightarrow [1, p]$ is the *Firing Constraint*

$\delta: [1, o \cdot a] \rightarrow ([1, p]_\perp \overset{strict}{\underset{1-1}{\longrightarrow}} [1, p]_\perp)$ is the *Transition Binding*.

(Notes: $\delta(n)$ needs only to be defined over $[1, \tau(n)]_\perp$)

Example: The sample F-Net from the last chapter can be represented formally as an order 4 (or greater) F-Net, $f_{example1} = \langle \Sigma, V, O, I \rangle$, where

$V = \{\text{i}, \text{hold1}, \text{hold2}, \text{sum}, \text{ans}\}$

$O = \{\text{oinit}, \text{opart}, \text{ored}, \text{ofin}\}$ where

$\text{oinit} \equiv \langle 2, \{\}, \{1,2\}, \{(1,1)\}, \phi_{init} \rangle$ where $\phi_{init}() \equiv (0, 51, 1, 1)$

$\text{opart} \equiv \langle 2, \{1\}, \{1,2\}, \{(1,2), (2,1)\}, \phi_{part} \rangle$

[2] In fact, a cleaner but more verbose definition would make O a sequence rather than a set, and the opcode an index into it. The existing definition will work under the assumption that firing functions are actually representations of functions.

where $\phi_{part}(n) \equiv (n-1,(n-1)^2,\begin{Bmatrix} 1 \text{ if } n=2 \\ 2 \text{ otherwise} \end{Bmatrix},1)$

$\textbf{ored} \equiv \langle 2,\{1,2\},\{2\},\{(1,1),(2,1)\},\phi_{red} \rangle$

where $\phi_{red}(n,m) \equiv (n+m,1,1)$

$\textbf{ofin} \equiv \langle 5,\{4\},\{5\},\{(1,1),(2,1),(3,1),(4,1),(5,1)\},\phi_{fin} \rangle$

where $\phi_{fin}(n) = (\sqrt{n},1,1,1,1,1)$

$I = \{\textbf{init},\textbf{part1},\textbf{part2},\textbf{red1},\textbf{red2},\textbf{fin}\}$ where

$\textbf{init} \equiv \langle \textbf{oinit}, \{(1,\textbf{sum}),(2,\textbf{i})\},$

$\{(1,1),(2,1)\},\{(1,\{(1,2)\}),(2,\{(1,2)\})\} \rangle$

$\textbf{part1} \equiv \langle \textbf{opar},\{(1,\textbf{i}),(2,\textbf{hold1})\},$

$\{(1,2),(2,1)\},\{(1,\{(1,3),(2,2)\}),(2,\{(1,2)\})\} \rangle$

$\textbf{part2} \equiv \langle \textbf{opar},\{(1,\textbf{i}),(2,\textbf{hold2})\},$

$\{(1,2),(2,1)\},\{(1,\{(1,3),(2,2)\}),(2,\{(1,2)\})\} \rangle$

$\textbf{red1} \equiv \langle \textbf{ored},\{(1,\textbf{hold1}),(2,\textbf{sum})\},$

$\{(1,2),(2,2)\},\{(1,\{(1,\{(1,1)\}),(2,\{(1,2)\})\} \rangle$

$\textbf{red2} \equiv \langle \textbf{ored},\{(1,\textbf{hold2}),(2,\textbf{sum})\},$

$\{(1,2),(2,2)\},\{(1,\{(1,\{(1,1)\}),(2,\{(1,2)\})\} \rangle$

$\textbf{fin} \equiv \langle \textbf{ofin},\{(1,\textbf{i}),(2,\textbf{hold1}),(3,\textbf{hold2}),(4,\textbf{sum}),(5,\textbf{ans})\},$

$\{(1,3),(2,1),(3,1),(4,2),(5,1)\},$

$\{(1,\{(1,3)\}),(2,\{(1,1)\}),(3,\{(1,1)\}),(4,\{(1,2)\}),(5,\{(1,2)\})\} \rangle$

4.3. Semantics

An F-Net computation will be described as a partial ordering, called an Execution Graph, containing two kinds of nodes: Event (E) nodes, which represent instruction firings (i.e. executions), and Variable Content (C) nodes, which represent the control and data state associated with a variable between instruction accesses. Arcs connect each C node to the E node representing the instruction firing which accesses the variable in that state, and connect each E node to a set of C nodes representing those same variables in their new (possibly un-modified) state. Since each C node will have at most one in-arc and one out-arc, the graph obtained by deleting all C nodes will also be a partial ordering. An execution graph illustrates how each instruction execution maps the old data states of each of its associated variables to new data and control states, or alternately, how each variable provides a means of communication and control between instructions, and is similar to the unrolling of a Petri net.

The set of possible execution graphs which correspond to a particular F-Net is presented by characterizing its members in two stages. First, the general form of an execution graph is given, then a set of axioms is provided which constrains the elements to execution graphs corresponding to the particular F-Net.

4.3.1. Form of an Execution Graph

Define an *execution graph* for an F-Net $f=\langle\Sigma, V, O, I\rangle$ with input $\iota: f\cdot V \rightarrow f\cdot\Sigma$ as a 6-tuple $x_{f,\iota}=\langle E, C, B, A, \hat{\sigma}, \bar{\sigma}\rangle$ where

E is a set of *firing events*

C is a set of *variable contents*

$B \subset C \times E$ is a set of *before arcs*

$A \subset E \times C$ is a set of *after arcs*

$\hat{\sigma}: E \rightarrow f\cdot I$ is an *instruction name labeling*

$\bar{\sigma}: C \rightarrow f\cdot V$ is a *variable name labeling*

such that functions

$\dot{\sigma}: C \rightarrow [1,p]_\perp$ is a *control state labeling*

$\ddot{\sigma}: C \rightarrow f \cdot \Sigma_\perp$ called a *data state labeling*

exist, and Semantic Axioms 1 through 6, described below, hold.

An execution is represented graphically with vertices $E \bigcup C$ and edges $B \bigcup A$. The vertices are labeled according to the values of their σ functions as follows:

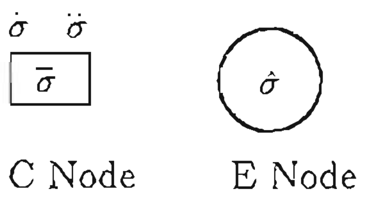


Figure 4.1 Node Labels for Execution Graph

See Figure 4.2 for parts of a possible execution graph for the sample F-Net.

4.3.2. Axioms Constraining Execution Graphs

In addition to Table 4.2, the following shorthand will be used in this section:

Define *the initial elements of C*, C_0, as those elements which precede all elements of C having the same name, i.e.

$$C_0 \equiv \{c_0 \in C \mid c \in C \wedge \bar{c} = \bar{c}_0 \Rightarrow c_0 \sim> c\}$$

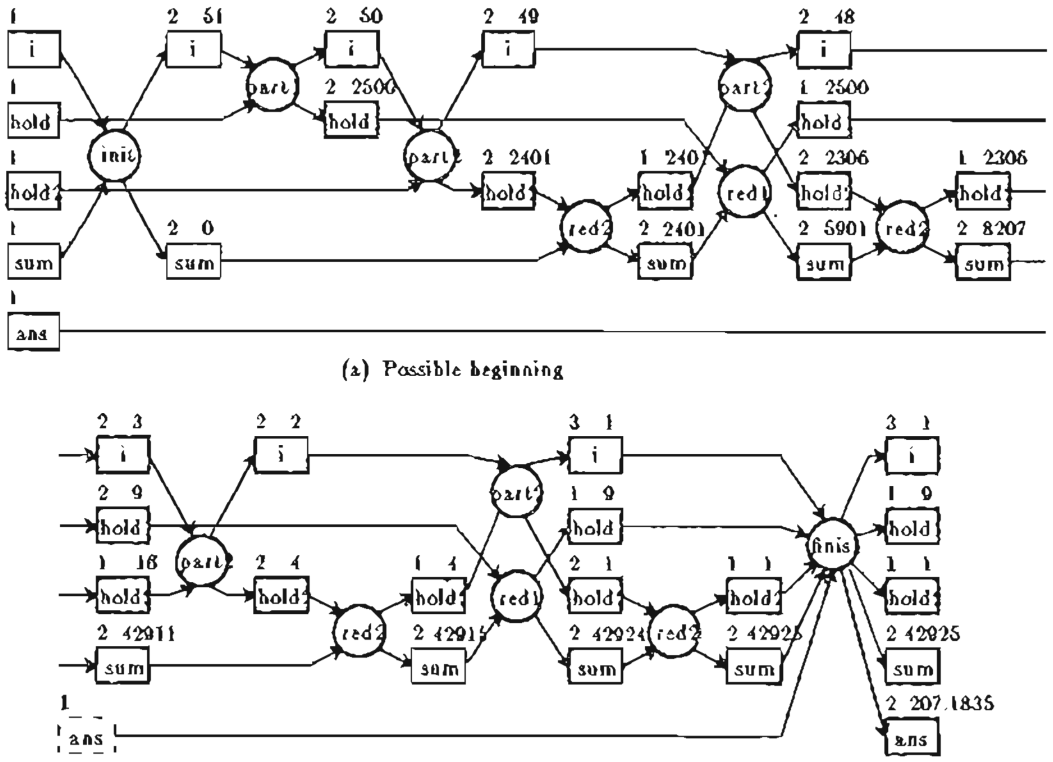
Axiom 1: Initial Conditions

Each variable has an initial C node named for it, which has a control state labeling of 1 and a data state labeling of ι.

$$\forall v \in V \exists c_0 \in C_0 . \bar{c}_0 = v \wedge \dot{c}_0 = 1 \wedge \ddot{c}_0 = \iota(v)$$

Axiom 2: Atomicity

A C node has at most one predecessor and one successor, signifying that it can be the result of at most one instruction execution and can be sensed by at most one instruc-



(a) Possible beginning

(b) Possible Ending

Figure 4.2 One Execution Graph for the Sample F-Net

Notation	Description	Meaning
$\hat{e}$	Instruction Label	$\hat{\sigma}(e)$
$\bar{c}$	Variable label	$\bar{\sigma}(c)$
$\dot{c}$	Control state	$\dot{\sigma}(c)$
$\ddot{c}$	Data state	$\ddot{\sigma}(c)$
$\alpha(L)$	Image of set L	$\{\alpha(l).l\in L\}$
$\alpha((y_i))$	Image of Seq.	(z_i) where $z_i=\alpha(y_i)$
$X\overset{m}{\leftrightarrow}Y$	m bijectively maps X to Y	$m(X)=Y \wedge m^{-1}(Y)=X$ where $m:X'\rightarrow Y'$, $X\subset X'$, $Y\subset Y'$
x^*	Successors	$\{y \mid (x,y)\in B\bigcup A\}$
$^*y$	Predecessors	$\{x \mid (x,y)\in B\bigcup A\}$
$x\sim>z$	Precedes	$(\exists y.(x,y)\in B\bigcup A \wedge y\sim>z)\vee x=z$ where $x,z\in C\bigcup E$
$L^>$	Set L as ascending sequence	(l_j) where $l_j\in L \wedge j<k==>l_j<l_k$
$L^{\sim>}$	Set L ordered by $\sim>$	
$x\vdash y$	x determines y	The axioms and x uniquely determine y

Table 4.2. Additional Notation

tion execution.

$$\forall c \in C.\ |c^*| \leq 1 \wedge |{}^*c| \leq 1$$

Axiom 3: Firing

Structure: An E node for an instruction has predecessor and successor C nodes which correspond exactly to the variables to which the arguments of the instruction are bound.

$$\forall e \in E.\quad {}^*e \overset{\bar{\sigma}}{\longleftrightarrow} \hat{e} \cdot \beta([1, \hat{e} \cdot o \cdot a])$$
$$\wedge\ e^* \overset{\bar{\sigma}}{\longleftrightarrow} \hat{e} \cdot \beta([1, \hat{e} \cdot o \cdot a])$$

Condition: The control states for the predecessors of an E node must correspond exactly to the firing constraint of the instruction represented by the E node.

$$\forall e \in E. c \in {}^*e \Rightarrow \dot{c} = \hat{e} \cdot \gamma(\hat{e} \cdot \beta^{-1}(\bar{c}))$$

Result: The firing function dictates the new data state of each of the instruction's written arguments and a transition for each of the instruction's arguments, based only on the old data state of the read arguments. The control state of each new variable content node is obtained by mapping its transition through its corresponding transition binding.

$$\forall e \in E. \hat{e} \cdot o \cdot \phi(\ddot{r}_1, \ddot{r}_2, \cdots) = (\ddot{w}_1, \ddot{w}_2, \cdots, t_1, t_2, \cdots, t_{\hat{e} \cdot o \cdot a}) \wedge$$
$$(\forall k \in [1, \hat{e} \cdot o \cdot a]. \dot{g}_k = \hat{e} \cdot \delta(k)(t_k))$$
$$\textit{where } r_j \in {}^*e,\ w_j, g_j \in e^*,\ (\bar{r}_j) = \hat{e} \cdot \beta(\hat{e} \cdot o \cdot R^<),$$
$$(\bar{g}_j) = \hat{e} \cdot \beta([1, \hat{e} \cdot o \cdot a]^<),$$
$$(\bar{w}_j) = \hat{e} \cdot \beta(\hat{e} \cdot o \cdot W^<)$$

Note: The sequences (r_j), (w_j), and (g_j) defined in the *where* clause are the predecessor nodes corresponding to read arguments, the successor nodes corresponding to write arguments, and all of the successor nodes, respectively. Axiom 3-structure ensures that there is exactly one sequence (c_j) of predecessor (or successor) nodes such that $(\tilde{c}_j)=\acute{e}\cdot\beta([1,\hat{e}\cdot o\cdot a]^{<})$. The three sequences in the *where* clause are subsequences of such a sequence, and so are well defined.

Axiom 4: Non-interference

If a variable is not a write variable for an instruction, then an execution of that instruction will not affect the data state of the variable.

$$\forall e\in E,\ m\in[1,\acute{e}\cdot o\cdot a]\backslash\hat{e}\cdot o\cdot W.$$
$$c\in{}^{*}e \wedge c'\in e^{*} \wedge \bar{c}=\bar{c}'=\acute{e}\cdot\beta(m) \Rightarrow \ddot{c}=\ddot{c}'$$

Axiom 5: Liveness

If an instruction can fire, it (or another instruction connected to some of its variables) must fire.

$$(\exists i\in I, C'\subseteq C.\ C'\overset{\bar{\sigma}}{\leftrightarrow} i\cdot\beta([1,i\cdot o\cdot a]) \wedge (\forall c\in C'.\ \dot{c}=i\cdot\gamma(i\cdot\beta^{-1}(\bar{c}))))$$
$$\Rightarrow \bigcup\{c^{*}\,|\,c\in C'\}\neq\emptyset$$

Axiom 6: Time Consistency

The execution graph will be acyclic:

$$(x,y\in C\bigcup E \wedge x\neq y \wedge x\sim>y)\Rightarrow not(y\sim>x)$$

In general, several execution graphs will satisfy the axioms for a particular F-Net and input.

4.4. Execution Graphs as Partial Orders

The theorems in this section will demonstrate that an execution graph is a partial ordering, that each variable is represented by exactly one bottom element within the partial ordering, and that the C nodes representing any variable are totally ordered within the partial order.

Theorem 1:

$\rightsquigarrow$ partially orders the elements of $E\bigcup C$.

Proof of Theorem 1:

Reflexivity and transitivity are obvious from the definition of $\rightsquigarrow$. Axiom 6 gives asymmetry.∎

Theorem 2:

There is exactly one element of C_0 representing each variable in the F-Net—i.e.

$$C_0 \overset{\bar{\sigma}}{\leftrightarrow} V$$

Proof of Theorem 2:

Axiom 1 gives $\bar{C_0}=V$. Let $\exists c,c'\in C_0.\bar{c}=\bar{c'}$. By definition of C_0, $\bar{c}\rightsquigarrow\bar{c}' \wedge \bar{c}'\rightsquigarrow\bar{c}$, but by Axiom 6, $\rightsquigarrow$ is asymmetric. So $c=c' \Rightarrow \bar{\sigma}:C_0 \rightarrow V$ is 1-1.∎

Define $C_v \equiv \{c'\in C.\bar{c}=v\}$.

i.e. C_v is the set of all C nodes labeled v.

Define $age:C\rightarrow\mathbf{N}$ as $age(c) \equiv |\{c'\in C_{\bar{c}}.c'\rightsquigarrow c\}|$

i.e. $age(c)$ is the number of C nodes with the same name preceding c (including c itself).

Theorem 3:

All variable content nodes with a given name labeling are totally ordered by $\rightsquigarrow$ (i.e. form a chain in the partial ordering):

$$\forall v\in V, c,c'\in C_v.\,not(c\rightsquigarrow c') \Rightarrow c'\rightsquigarrow c$$

Proof of Theorem 3:

The proof will consist of showing that

$$C_v \overset{age}{\leftrightarrow} [1,|C_v|]$$

from which the proof of the theorem is obvious.

Define $B(c)\equiv\{c'\in C_{\bar{c}}.c'\neq c \wedge c'{\sim>}c \wedge (\forall c''\in C_{\bar{c}}.c''{\sim>}c \Rightarrow c''{\sim>}c')\}$.

That is, $B(c)$ is the set of "closest" members of $C_{\bar{c}}$ which precede c, so

$$age(c)=1+\sum_{c'\in B(c)} age(c').$$

Let $c',c''\in B(c)$. Then from definition of $B(c)$ and Axiom 2,

$$c'{\sim>}c \Rightarrow |c'^{*}|\neq 0 \Rightarrow |c'^{*}|=1,$$

and the same is true for c''. Let $c'^{*}=\{e'\}$, $c''^{*}=\{e''\}$. $|e'^{*}\cap C_{\bar{c}}|=|e''^{*}\cap C_{\bar{c}}|=1$ from Axiom 3-structure. In each case, that element must be c. From Axiom 2, $e'=e''$, and since $|{}^{*}e'\cap C_{\bar{c}}|=1$, $c'=c''$. Thus, $B(c)=\{c'\}$, and $age(c')=age(c)-1$, so in general,

$$\forall v\in V, n>1. |\{c\in C_v.age(c)=n\}|\leq|\{c\in C_v.age(c)=n-1\}|.$$

The proof is finished by observing that

$$|\{c\in C_v.age(c)=1\}|=1$$

by Axiom 1 and definition of C_0. •

4.5. Execution Graphs as Computations

Define $c_{\sim>}\equiv\{c'\in C.c'{\sim>}c \wedge c'\neq c\}$

Theorem 4:

The control state and data state labelings (functions $\dot{\sigma}$ and $\ddot{\sigma}$) for an execution graph are unique: i.e.

$$\vdash \dot{\sigma},\ddot{\sigma}$$

Proof of Theorem 4:

Induction over partial order $C^{\sim>}$.

Base Case: $c\in C_0 \vdash \dot{c},\ddot{c}$

From axiom 1, $\dot{c}=1 \wedge \ddot{c}=\iota(\bar{c})$

Inductive Case: $\forall c \in C.((c' \in c_{\leadsto} \vdash \ddot{c}') \vdash \dot{c}, \ddot{c})$.

$c' \in c_{\leadsto} \Rightarrow |{}^{*}c|=1$ from axiom 2. Let ${}^{*}c=\{e\}$, and let (r_i) be the sequence of e's predecessors corresponding to its read arguments, i.e.

$$(\bar{r}_i)=\hat{e}\cdot\beta(\hat{e}\cdot o\cdot R^{<}) \text{ where } r_i \in {}^{*}e.$$

By construction, $\forall i.r_i \in c_{\leadsto}$, so by the inductive assumption, $\vdash \ddot{r}_i$. Let

$$(n_i)\equiv\hat{e}\cdot o\cdot\phi(\ddot{r}_1,\ddot{r}_2,\cdots,\ddot{r}_l) \qquad \text{(i.e. the result vector from the instruction)}$$

and $m\equiv\hat{e}\cdot\beta^{-1}(\bar{c})$ (i.e. the argument represented by c).

Axioms 3-result and 4 dictate

$$\dot{c}=\hat{e}\cdot\gamma(m)(n_{m+|\hat{e}\cdot o\cdot W|}).$$

and

$$\ddot{c}=\begin{cases} n_k & \text{if } m=(\hat{e}\cdot o\cdot W^{<})_k \\ \ddot{c}'' & \text{if } m\notin\hat{e}\cdot o\cdot W \end{cases}$$

where c'' is the member of ${}^{*}e$ such that $\bar{c}=\bar{c}''$. ∎

The fact that $\dot{\sigma}$ and $\ddot{\sigma}$ are completely determined by an execution graph explains why they are not taken to be part of the definition of the graph, but it also illustrates that each C node represents the results of a function evaluation, perhaps on its way to be used as an argument to another function. To get a better feeling for the function evaluation taking place, the $\ddot{\sigma}$ labeling can be interpreted slightly differently, as the function evaluation represented by the node rather than the result of that evaluation, simply by leaving the ϕ functions unreduced while following the procedure used in the proof of Theorem 4.

Define the *output* of a execution graph $x_{f,t}$ relative to variable $v \in V$ and control state $n \in [1,p]$, denoted $output(x_{f,t},v,n)$, as the sequence

$$\ddot{\sigma}(\{c \in x_{f,\iota}\cdot C_s . c = n\}^{\sim>})$$

(Corollary 3.1 and Theorem 4 prove that this is well defined.)

Define an F-Net as being *determinate* with respect to variable $v \in V$ and control state $n \in [1,p]$, if and only if

$$\forall \iota . output(x_{f,\iota}, v, n) = output(x'_{f,\iota}, v, n)$$

(i.e. if the output sequence for control state n of variable v is dependent only on the input for a given F-Net).

4.6. Tracing an Execution

How much (or little) information is required, in addition to the F-Net itself, to completely determine an execution graph for that F-Net (up to an isomorphism)? This question is important for debugging non-deterministic programs, for it determines the amount of data that must be logged during an execution in order to reconstruct "what happened" during that execution.

Define, for $i, i' \in I$,

$$shared(i,i') \equiv i \cdot \beta([1, i \cdot o \cdot a]) \cap i' \cdot \beta([1, i' \cdot o \cdot a])$$

(i.e. the set of variables to which both i and i' are bound).

Define the *contends* relation $<>$ as

$$i <> i' \equiv i \neq i' \wedge shared(i,i') \neq \emptyset$$
$$\wedge\ \forall v \in shared(i,i') . i \cdot b(i \cdot \beta^{-1}(v)) = i' \cdot b(i' \cdot \beta^{-1}(v)))\}$$

i.e. two instructions contend whenever they have at least one variable in common, and for all variables which they have in common, their firing is constrained to the same control state of that variable.

Put another way, two instructions contend if the states of the variables which they share do not dictate which should execute next. Since they do share variables, the E nodes representing their executions will be related by $\rightsquigarrow$, so the order in which they execute will affect the topology of the execution graph. This suggests (and the remainder of this chapter will prove) that contending instructions are the only source of non-determinism in an F-Net. We now add *instrumentation* to an F-Net to capture the order in which contending instructions fire, and thus the non-deterministic choices made during an F-Net execution. This is accomplished by coloring the instructions of the F-Net such that contending instructions always have differing colors, then recording the color of each contending instruction every time it fires. This log of colors does not need to be global—it is only necessary that any two instructions which contend use a common log. Since contending instructions already have some common variables by definition, these variables provide a handy (and local) site to store the logs. It is not necessary to assign a log to each shared variable, but to at least one variable shared by the contending instructions.

Define an *Instrumented F-Net* as a 6-tuple $\tilde{f}=\langle \Sigma, V, O, I, color, logsel \rangle$ where

$f=\langle \Sigma, V, O, I \rangle$ is an F-Net

$color: I \rightarrow \mathbf{N}$ is an *Instruction Coloring*

$logsel: I \rightarrow \mathbf{P}(V)$ is a *Log Selector*

such that

$$(i <> i') \Rightarrow color(i) \neq color(i') \wedge logsel(i) \cap logsel(i') \cap shared(i,i') \neq \emptyset$$

$$not(i <> i') \Rightarrow logsel(i) = \emptyset$$

i.e. for any two instructions which contend, the colors assigned to the instructions are different, and at least one of their shared variables belongs to the log selector of each instruction.

Example:

To better convey the points of this section, we now leave the previous example behind and refer to the F-Net shown in Figure 4.3. It will not be necessary to detail the firing functions for the individual instructions. In this F-net, $B<>C$ and $B<>D$ are the only instructions which contend. One possible instrumentation for that F-Net is

$color \equiv \{(A,0),(B,0),(C,1),(D,1),(F,0)\}$ and

$logsel \equiv \{(A,\{\}),(B,\{L,M\}),(C,\{M\}),(D,\{L\}),(F,\{\})\}$

Define a *Traced Execution Graph* of instrumented F-Net $\tilde{f}=\langle\Sigma,V,O,I,color,logsel\rangle$ with input ι as a 7-tuple $\bar{x}_{\tilde{f},\iota}=\langle E,C,B,A,\hat{\sigma},\bar{\sigma},logent\rangle$ where

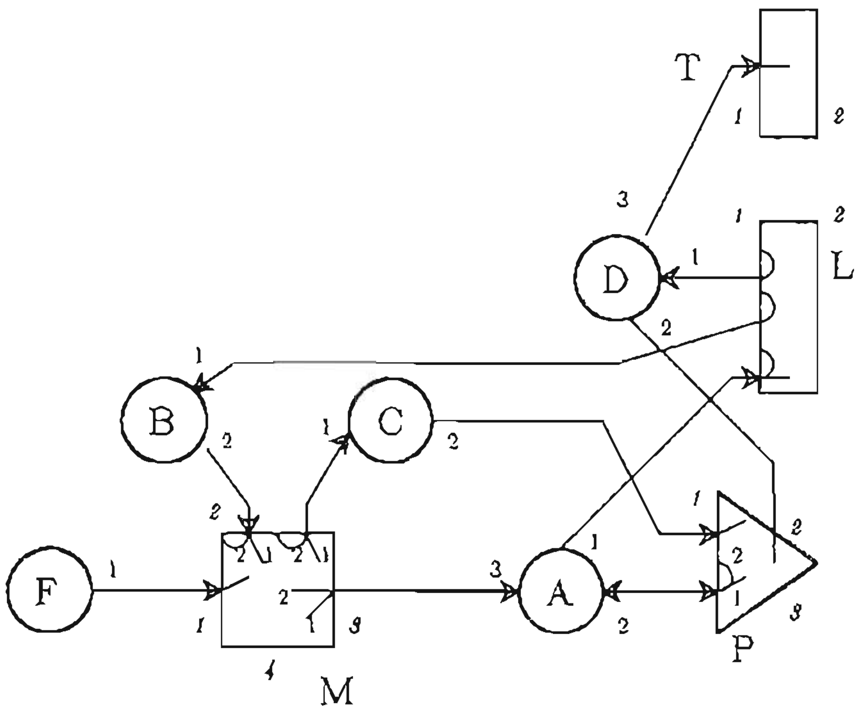


Figure 4.3 A New Sample F-Net

$x_{f,\iota} = \langle E, C, B, A, \acute{\sigma}, \bar{\sigma} \rangle$ is an execution graph for F-Net $f = \langle \Sigma, V, O, I \rangle$ with input ι

$logent : C \rightarrow \mathbb{N}$ is the *Log Entry*

such that

$$\forall e \in E \, \forall c \in {}^{*}e . \bar{c} \in logsel(\acute{e}) \Rightarrow logent(c) = color(\acute{e})$$

i.e. each time a contending instruction fires, the color of the instruction is logged to all of its predecessor nodes which correspond to its log selectors.

Define the *Log* of traced execution graph $\tilde{x}_{\tilde{f},\iota}$ for variable $v \in V$, denoted $log(\tilde{x}_{\tilde{f},\iota}, v)$, as

$$logent(\{c \in C_v . c^{*} = \{e\} \wedge v \in logsel(\acute{e})\}^{\sim>})$$

where

$$\tilde{f} = \langle \Sigma, V, O, I, color, logsel \rangle$$

$$\tilde{x}_{\tilde{f},\iota} = \langle E, C, B, A, \acute{\sigma}, \bar{\sigma}, logent \rangle$$

i.e. the log for a variable is the sequence of log entries assigned to the C nodes named for the variable, omitting those that do not immediately precede contending instructions.

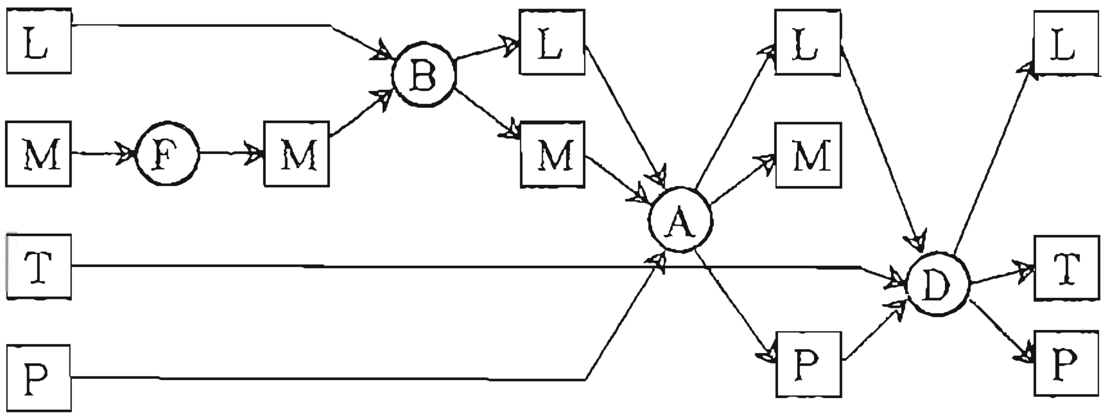


Figure 4.4. An Execution Graph for F-Net in Figure 4.3

Example:

If the execution graph in Figure 4.4 is a traced execution graph of the instrumented F-Net given in the last example, then

$\log(\tilde{x}_{f,t}, L) = 0, 1$

$\log(\tilde{x}_{f,t}, M) = 0$

All other logs are empty •

Minimizing the range of *color* will therefore minimize the size (i.e. number of bits) for a log entry, and minimizing the range of *logsel* will minimize the number of logs to which each log entry is recorded.

4.7. Execution Graphs with Identical Logs are Isomorphic

Definition: Prefix subgraph

Let x be an execution graph. y is a *prefix subgraph* of x if y is a subgraph of x, if every predecessor in x of an element of y is also in y (as is the arc between them), and if every successor to an E node in y is also in y—i.e. the following conditions hold:

(1) $c \in y \cdot C \Rightarrow c \in x \cdot C$

(2) $e \in y \cdot E \Rightarrow e \in x \cdot E$

(3) $(c \in y \cdot C \wedge (e,c) \in x \cdot A) \Leftrightarrow (e \in y \cdot E \wedge (e,c) \in y \cdot A)$

(4) $(e \in y \cdot E \wedge (c,e) \in x \cdot B) \Leftrightarrow (c \in y \cdot C \wedge (c,e) \in y \cdot B)$

(5) $(e \in y \cdot E \wedge (e,c) \in x \cdot A \Rightarrow c \in y \cdot C$

Note that a prefix subgraph is often not a legal execution graph because it does not adhere to Axiom 5-liveness.

Definition: Graph Isomorphism

Execution graphs (or prefix subgraphs) x and x' are isomorphic (denoted $x \cong x'$) iff there exist bijections $\tilde{c}: x\cdot C \leftrightarrow x'\cdot C$ and $\tilde{e}: x\cdot E \leftrightarrow x'\cdot E$ such that

$$(c,e) \in x\cdot B \Leftrightarrow (\tilde{c}(c), \tilde{e}(e)) \in x'\cdot B$$

$$(e,c) \in x\cdot A \Leftrightarrow (\tilde{e}(e), \tilde{c}(c)) \in x'\cdot A$$

$$\bar{\sigma}(c) = \bar{\sigma}(\tilde{c}(c))$$

$$\hat{\sigma}(e) = \hat{\sigma}(\tilde{e}(e))$$

From the proof of Theorem 4, it also follows that

$$\dot{\sigma}(c) = \dot{\sigma}(\tilde{c}(c))$$

$$\ddot{\sigma}(c) = \ddot{\sigma}(\tilde{c}(c))$$

The rest of the chapter will be devoted to proving the following theorem:

Theorem 5:

If $\tilde{x}_{\tilde{f},\iota} = \langle E, C, B, A, \hat{\sigma}, \bar{\sigma}, logent \rangle$ and $\tilde{x}'_{\tilde{f},\iota} = \langle E', C', B', A', \hat{\sigma}', \bar{\sigma}', logent' \rangle$ are traced execution graphs of instrumented F-Net $\tilde{f} = \langle \Sigma, V, O, I, color, logsel \rangle$ with input ι such that $\forall s \in S. \log(\tilde{x}_{\tilde{f},\iota}, v) = \log(\tilde{x}'_{\tilde{f},\iota}, v)$ then execution graphs $x_{f,\iota} = \langle E, C, B, A, \hat{\sigma}, \bar{\sigma} \rangle$ and $x'_{f,\iota} = \langle E', C', B', A', \hat{\sigma}', \bar{\sigma}' \rangle$ of $f = \langle \Sigma, V, O, I \rangle$ are isomorphic.

That is, in addition to the original input and instrumented F-Net, only the log associated with each variable is needed to uniquely determine the execution graph. The proof will be presented "top-down" to give the reader a better bearing on where it is all leading, and is based on an induction over the partial ordering represented by the execution graphs.

Proof of Theorem 5

Base case:

The prefix subgraph consisting only of the C_0 elements of x is isomorphic to the prefix subgraph consisting only of the C_0 elements of x'. This follows immediately from the definition of C_0, Axiom 1, and the definition of a prefix subgraph.

Inductive step:

Suppose that there is a prefix subgraph of $x_{f,t}$ (call it y) which is isomorphic to a prefix subgraph of $x'_{f,t}$ (call it y'). If either execution graph (say x, WLOG) has an element not in its prefix subgraph (y), then pick a least such element by partial ordering $\rightsquigarrow$, and call it e_x. (It must be a member of E from the definition of prefix subgraph.) We will show that the other execution graph (x') has an element $e_{x'}$ not in its prefix subgraph (y') such that adding e_x, its before arcs, after arcs, and successors to y is a prefix subgraph which is isomorphic to that obtained by adding $e_{x'}$, its before arcs, after arcs, and successors to y'.

The result of this induction is that given any two traced execution graphs with identical logs for all variables, all of both graphs can be pulled into y and y', so the execution graphs themselves must be isomorphic.

Proof of the inductive step:

Construct y, y', and e_x as indicated in the above proof statement.

Define

$$P_x \equiv {}^{*}e_x$$

All members of P_x must be members of y, so by the inductive assumption, there exists a set $P_{x'}$ in y' such that

$$P_{x'} \cong P_x$$

Consider the successors of $P_{x'}$:

$$E'_{x'} \equiv \bigcup_{c_x \in P_{x'}} c_{x'}{}^*$$

$E'_{x'}$ cannot be empty, since we know from execution graph x that there exists an instruction ($\hat{e}_x$) which can fire based on the control states of the elements of P_x, so by Axiom 5-Liveness and isomorphism, one of the elements of $P_{x'}$ must have a successor.

Pick a least element of $E'_{x'}$ and call it $e_{x'}$. In the next section, Lemma 5.5 will show that $\hat{e}_{x'} = \hat{e}_x$, and from Lemma 5.2 it will follow that ${}^*e_{x'} \cong {}^*e_x$, so $E' = \{e_x'\}$. Axiom 3-Structure then gives that $e_{x'}{}^*$ and $e_x{}^*$ have the same names, control states, and data states.

4.8. Toward Proving $\hat{e}_x = \hat{e}_{x'}$

Lemma 5.1: $shared(\hat{e}_x, \hat{e}_{x'}) \neq \emptyset$

Proof:

$P_x \cong P_{x'}$	(By construction)
$\Rightarrow \bar{\sigma}(P_x) = \bar{\sigma}(P_{x'})$	(Mapping over like sets)
${}^*e_x = P_x$	(By construction)
$\Rightarrow \bar{\sigma}({}^*e_x) = \bar{\sigma}(P_x)$	(Mapping over like sets)
${}^*e_{x'} \cap P_{x'} \neq \emptyset$	(By construction)
$\Rightarrow \bar{\sigma}({}^*e_{x'}) \cap \bar{\sigma}(P_{x'}) \neq \emptyset$	
$\Rightarrow \bar{\sigma}({}^*e_{x'}) \cap \bar{\sigma}({}^*e_x) \neq \emptyset$	(Substitution)
$\Rightarrow \hat{e}_x \cdot \beta(\|1, \hat{e}_x \cdot o \cdot a\|) \cap \hat{e}_{x'} \cdot \beta(\|1, \hat{e}_{x'} \cdot o \cdot a\|) \neq \emptyset$	(Axiom 3-struct)
$\Rightarrow shared(\hat{e}_x, \hat{e}_{x'}) \neq \emptyset$	(Def. of shared)

Lemma 5.2: $c_x \in^* e_x \wedge c_{x'} \in^* e_{x'} \wedge \bar{c}_x = \bar{c}_{x'} \Rightarrow c_x \cong c_{x'}$

Proof:

Let $c_x \in^* e_x$ and $c'_{x'} \in^* e_{x'}$ such that $\bar{c}_x = \bar{c}'_{x'}$. Since $c_x \in P_x$, there must be a $c_{x'} \in P_{x'}$ such that $c_x \cong c_{x'}$, and therefore $\bar{c}_x = \bar{c}_{x'}$. We will prove that $c_{x'} = c'_{x'}$, thus proving the Lemma.

Suppose that $c'_{x'} \neq c_{x'}$.

From Theorem 3, all elements of C with the same name form a chain in the partial order, so either (a) $c'_{x'} \rightsquigarrow c_{x'}$. or (b) $c_{x'} \rightsquigarrow c'_{x'}$

(a) $c'_{x'} \in^* e_{x'}$, so by Axiom 2-Atomicity, $c'_{x'}{}^* = \{e_{x'}\}$. From this, the definition of $\rightsquigarrow$, and the fact that $c'_{x'} \rightsquigarrow c_{x'}$, we get $e_{x'} \rightsquigarrow c_{x'}$. But $c_{x'}$ is a member of y', so by the definition of prefix subgraph, $e_{x'}$ must also be. $\Rightarrow\Leftarrow$.

(b) Let $c_{x'} = \{e'_{x'}\}$. (It must have exactly one element, from Axiom 2-Atomicity and the fact that it precedes other elements.) But from $c_{x'} \rightsquigarrow c'_{x'}$ and the construction of $c'_{x'}$, it follows that $c_{x'} \rightsquigarrow e'_{x'} \rightsquigarrow c'_{x'} \rightsquigarrow e_{x'}$. Thus, $e_{x'}$ is not a least element having a predecessor in $P_{x'}$. $\Rightarrow\Leftarrow$.

So $c'_{x'} = c_{x'}$, and therefore $c'_{x'} \cong c_x$. •

Lemma 5.3

$$\forall v \in shared(\hat{e}_x, \hat{e}_{x'}) \exists c_x \in^* e_x,\ c_{x'} \in^* e_{x'}. \bar{c}_x = \bar{c}_{x'} = v$$

Proof:

$v \in shared(\hat{e}_x, \hat{e}_{x'}) \Rightarrow v \in \hat{e}_x \cdot \beta([1, \hat{e}_x \cdot o \cdot a]) \cap \hat{e}_{x'} \cdot \beta([1, \hat{e}_{x'} \cdot o \cdot a])$ (Def. of shared)

$\Rightarrow \exists c_x \in^* e_x,\ c_{x'} \in^* e_{x'}. \bar{c}_x = \bar{c}_{x'} = v$ (Axiom 3-structure)•

Lemma 5.4

$\hat{e}_x <> \hat{e}_{x'}$

Proof of Lemma 5.4

$c_x \in^* e_x \wedge c_{x'} \in e_{x'} \wedge \bar{c}_x = \bar{c}_{x'} \Rightarrow c_x \cong c_{x'}$ (Lemma 5.2)

$\forall v \in shared(\hat{e}_x, \hat{e}_{x'}) \exists c_x \in^* e_x,\ c_{x'} \in^* e_{x'}. \bar{c}_x = \bar{c}_{x'} = v$ (Lemma 5.3)

$\Rightarrow \forall v \in shared(\hat{e}_x, \hat{e}_{x'}) \exists c_{x'} \in^* e_{x'}, c_x \in^* e_x . \bar{c}_{x'} = \bar{c}_x = s \wedge \dot{c}_{x'} = \dot{c}_x$ (Substitution)

$\forall e \in E. c \in^* e \Rightarrow \dot{c} = \hat{e} \cdot b(\hat{e} \cdot \beta^{-1}(\bar{c}))$ (Axiom 3-cond)

$\Rightarrow \forall v \in shared(\hat{e}_x, \hat{e}_{x'}). \hat{e}_x \cdot b(\hat{e}_x \cdot \beta^{-1}(v)) = \hat{e}_{x'} \cdot b(\hat{e}_{x'} \beta^{-1}(v))$ (Substitution)

$shared(\hat{e}_x, \hat{e}_{x'}) \neq \emptyset$ (Lemma 5.1)

$\Rightarrow \hat{e}_x <> \hat{e}_{x'}$ (Def of $<>$)•

Lemma 5.5

$\hat{e}_x = \hat{e}_{x'}$

Proof:

Suppose false: i.e.

$\hat{e}_x \neq \hat{e}_{x'}$

After adding the results of Lemma 5.4, the definition of an instrumented F-Net gives

$logsel(\hat{e}_x) \cap logsel(\hat{e}_{x'}) \cap shared(\hat{e}_x, \hat{e}_{x'}) \neq \emptyset$.

Let v be a member of that set. Then it must be a member of $shared(\hat{e}_x, \hat{e}_{x'})$, so by lemma 5.3,

$\exists c_x \in^* e_x, c_{x'} \in^* e_{x'}. \bar{c}_x = \bar{c}_{x'} = v$

From this, and the fact that $v \in logsel(\hat{e}_{x'}) \cap logsel(\hat{e}_x)$, and the definition of a traced execution graph,

$logent(c_x) = color(\hat{e}_x)$

$logent(c_{x'}) = color(\hat{e}_{x'})$

Since c_x and $c_{x'}$ are corresponding elements from isomorphic prefix subgraphs, and since the logs for the execution graphs are identical,

$$logent(c_{x'})=logent(c_x)$$

so

$$color(\hat{e}_{x'})=color(\hat{e}_x)$$

But the definition of an instrumented F-Net expressly requires that

$$\hat{e}_{x'}\neq\hat{e}_x \wedge \hat{e}_x<>\hat{e}_{x'}\Rightarrow color(\hat{e}_{x'})\neq color(\hat{e}_x)$$

Thus the contradiction, so $\hat{e}_{x'}=\hat{e}_x$.●

4.9. Conclusions

The previous two sections showed that, in general, only a small amount of information needs to be recorded during an execution to allow for the reconstruction of that execution, and that this information only needs to be recorded for instructions which contend. In the example given, this consisted of recording one bit of information whenever instructions C or D fired, and two bits whenever B fired. In addition, the recording is always performed to an uncontested site, namely a variable which is already accessed by the instruction performing the recording. It seems plausible that the space and time overhead for this recording will be small enough in the general case that the benefit gained by instrumenting every F-Net will not be negated by any significant loss in performance during its execution.

The fact that only the firings of contending instructions need to be recorded directly implies that if an F-Net has no such instructions, all executions of that F-Net will be deterministic: i.e. given the same input, all execution graphs will be isomorphic.

4.10. Final Note on the Effects of Order (Size of Control Domain)

An order-p F-Net is one in which each of its variables has a control domain of (at most) p elements (plus $\perp$). Variables in an order-1 F-Net can therefore provide no means to control the order in which instructions fire and thus no means to enforce communication between instructions. A legal execution for any order-1 F-Net could consist of a single instruction associated with each variable firing repeatedly forever (or until variables attain a control state of $\perp$). Order-1 F-Nets are therefore clearly of little use.

An F-Net of order 2 or higher can always be expressed as an F-Net of order 2 with the same number of instructions by modeling each n-control-state variable with $lg(n)$ 2-control-state variables. This provides n different ways of constraining the firing of the instruction. See Figure 4.5. By selectively producing transitions to these variables, any of these instructions can arbitrarily determine the next control state for each of the variables, thereby dictating which connectivity will be enabled next. This is exactly the behavior required for an n-control-state variable.

"Non-volatility" is preserved by this transformation. If an instruction in a high-order F-Net has non-volatile access to a variable, the instruction can always be modeled as performing a single transition to each of that variable's representatives in the order-2 F-Net with no write usage to any of them. If the high-order instruction does have write usage to

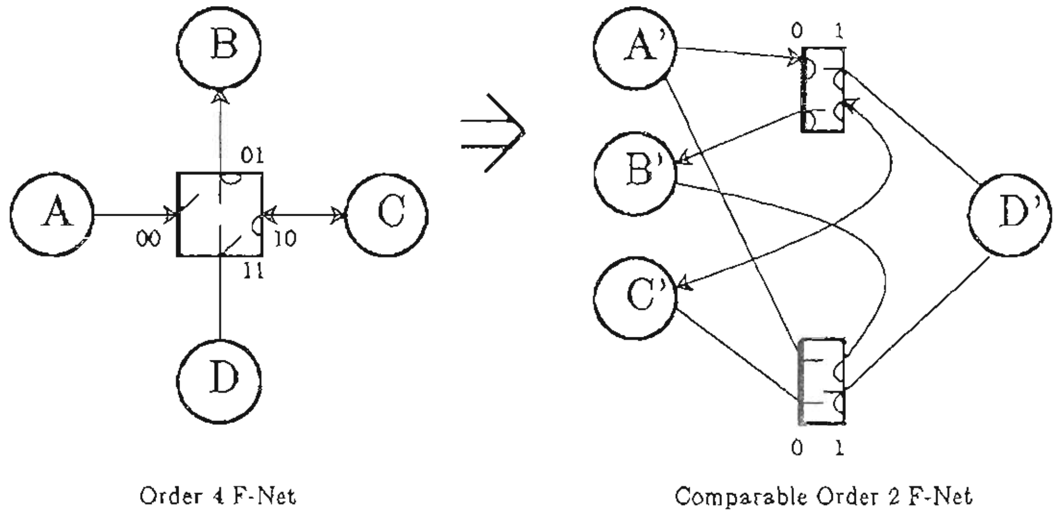


Figure 4.5. Comparable Order-4 and Order-2 F-Nets

the variable, or performs multiple transitions, it must have write usage to, or perform multiple transitions to, at least one of the representative variables in the order-2 F-Net.

Even though an order 2 F-Net can always be constructed to have the same behavior as a higher-order F-Net, the amount of high-level information about that behavior is greater in the higher-order F-Net.

(1) Two different instructions in a high-order F-Net which represent the same mapping from read arguments to write arguments and which perform the same number of transition to each argument can use the same operation. *After* translating to an order-2 F-Net, it may not be possible to use the same operation for both because the transitions performed by the operation may depend on the encoding of the variable's control domain.

(2) It is not possible to represent as many transition bindings in a low-order F-Net as in a higher-order one. For example, in Figure 4.5, it is apparent that instruction D will only make transitions to control states 01 or 10, enabling instructions B or C, but this is not apparent at all in the order-2 F-Net. The possible combinations of transitions have been hidden inside of the firing function of D'.

Additional justification for having nets of order higher than 2 will become clear when the model is extended to include hierarchy. For these reasons, low-order F-Nets will not be further considered in this thesis, and the term "F-Net" will subsequently refer to an order infinity F-Net—i.e. one in which the variables have as large of a control domain as needed. This is possible because control states to which there are no transitions or bindings have no effect on the semantics and are not shown in the graphical representation.

CHAPTER 5

Comparison with Other Models

Now that F-Nets have been formally defined, some comparisons can be made between F-Nets and some of the other models mentioned in Chapter 2.

5.1. Unity

Unity programs are very similar to order 1 F-Nets[1]—i.e. F-Nets with a control domain capable only of ensuring the atomicity of execution. Unity provides two extensions over these nets, however: (1) a strong notion of fairness, ensuring that no instruction firing will be delayed for more than a finite number of other instruction firings, and (2) an optional guard for each instruction which prevents it from having an effect when not satisfied. Unity-like guards could be simulated in order-1 F-Nets by including conditionals within operations and requiring arguments to have read usage whenever they have write usage. This latter restriction is necessary because every firing would be required to produce a new data state for these arguments, whether or not the simulated guard was satisfied.

On a more practical level, a Unity program, as presented by the authors, has relatively fine-grained processes and ether. The ability to collect ether addresses into larger structures is important to achieve the granularity in high-latency environments needed for portable parallel programs.

[1] Recall that the order of an F-Net is the maximum size of the control domains for its variables.

5.2. Petri Nets

An operation which has no arguments having read or write usage must have a firing function which returns constant transitions. An F-Net containing only operations of this kind can be modeled directly as a Petri net. Each element (other than $\perp$) of the control domain of each variable in the F-Net becomes a place in the Petri Net, and each instruction becomes a Petri-Net transition. Each argument in the F-Net becomes two arcs in the Petri Net—an input arc representing the firing constraint and an output arc representing the transition binding. The initial marking for the Petri Net consists of one token on each place which corresponds to an initial control state; see Figure 5.1.a.

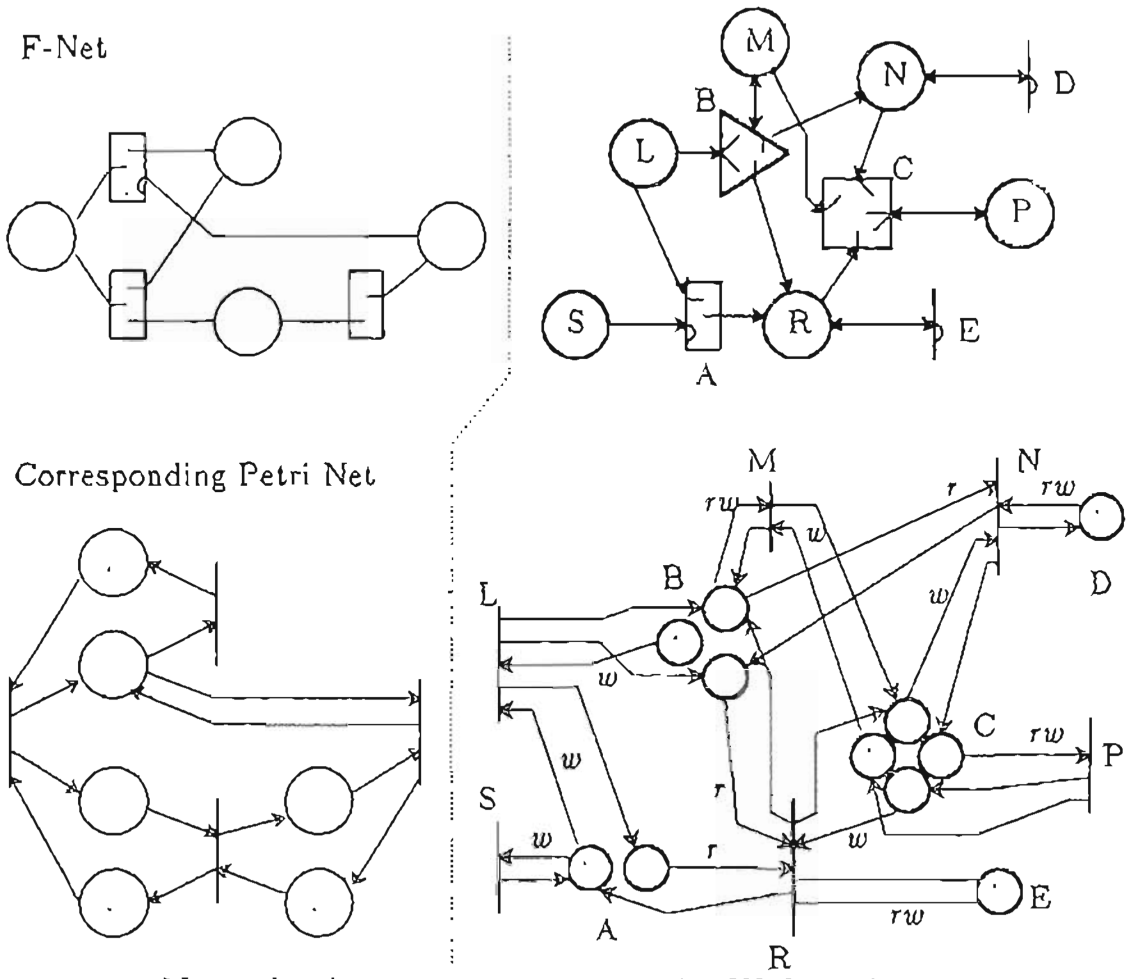
To model F-Nets more generally requires that Petri Net semantics be extended. In addition to the above translation (now with multiple output arcs for each argument, one for each transition binding), the following additions are required (see Figure 5.1.b for an example):

(1) The input arcs to the transitions in the Petri Net are colored (with chalk) red, white, or both, depending on whether the associated argument has read usage, write usage, or both. Arcs having neither are left uncolored.

(2) Places are labeled for the variable which they represent.

(3) Each token is extended to carry a data value, and is labeled indelibly with the name of the variable to which it belongs.

(4) When a transition fires, it takes one token from each input place. As it is taken, some of the chalk from the arc will smudge onto the token. (If the arc has both red and white, both colors will smudge.) Based only on the data associated with tokens which are smudged with red chalk, the transition determines new data values for tokens smudged with white, then determines an output arc for all tokens. The



a. No read-write usage b. With read-write usages

Figure 5.1. F-Nets Modeled as Petri Nets

output arc chosen must be connected to the appropriate variable (i.e. that for which the token is labeled), and the transition may determine that it will not replace the token at all if the token is smudged white or if there is more than one output place corresponding to its variable.[2] As a token is put on an output place, all smudges are cleaned off the token. Note that unlike conventional Petri Nets, a token is *not* added to each output place, but to at most one output place corresponding to each

[2] This latter case represents the $\perp$ control state.

m-variable.

Other extended versions of Petri Nets which include data transformations and timing of transitions have been proposed by other researchers, primarily to simulate hardware systems. While the F-Net model has intentionally avoided addressing timing constraints, we believe that it can address all other aspects of these models.

5.3. CCS

F-Nets and Milner's CCS have a great deal of similarity. Both use, as a basis, finite state machines. Events can occur only when the states of different machines occur in stated combinations. These events are atomic, and when they occur, they cause (or allow) a transition to the state of each of the machines involved.

The specifics of the models are different, however. In CCS, the events are communication, where no data transformation takes place, while in F-Nets, the events are instruction firings, which do transform data. In CCS, a state transition may involve a data transformation, and may be non-deterministic, while in F-Nets, the state transition itself does not transform data, and is deterministic. However, the state transition and data transformation in F-Nets depends on the event which causes the transformation and the data which that event accesses (unlike CCS where the transformation depends only upon the previous state of the machine plus non-determinism), and that event can be non-deterministically chosen in some cases.

From these comparisons, it seems clear that an F-Net can be constructed with identical behavior to any CCS program, by representing each CCS communication link with one or more F-Net instructions, as necessary to provide the required non-determinism. The possibility that some aspects CCS and F-Net theories could be merged could provide fertile ground for future research.

5.4. Functional Models

A primary difference between F-Nets and traditional functional models is its lack of single-assignment variables, and therefore lack of referential transparency in the general case. A limited amount of referential transparency can be obtained within an F-Net by considering each m-variable to be a set of functional variables, one for each element of the control domain, and ensuring that each instruction with write usage makes a transition to a "new" (say, numerically higher) control state. But any usable functional model must be able to create new contexts with new versions of single-assignment variables to avoid using up the supply. These contexts are typically created for each iteration of a loop, each invocation of a function, or by using a local assignment (`let ... in`) facility. In the absence of these contexts, the control state of an m-variable provides a method for explicitly managing the reuse of control states, rather than relegating the analysis of re-use to a smart compiler.

5.5. Guarded Commands

Assuming that each element of the control domain of each m-variable represents a predicate over the data state of that m-variable, and the control state represents one of those predicates which is asserted to be true, the firing constraint of each instruction can be regarded as a guard formed by the conjunction of these predicates. Any guarded command can therefore be modeled by expressing its guard in disjunctive normal form and creating a separate instruction (with the same operation) for each disjunct. This F-Net form of guarded commands clearly shows the relationships between the guards of different commands, both to the human in terms of the graphical form of the F-Net and to a scheduler when determining when a guard must be re-evaluated. A compiler for Unity could very well use such an F-Net as an intermediate form.

Similarly, the control states for a variable can be regarded as exception conditions, with the instructions constrained by each control state being the exception handlers.

5.6. Graphical Specification Languages

Common graphical specification languages either detail the possible data relationships between modules without defining the control relationships (e.g. dataflow diagrams or Entity-Relation diagrams), or they detail the control relationships without detailing the data relationships (e.g. Petri Nets). In F-Nets, all of the information about each instruction's behavior other than the actual mathematical transformation between input and output is shown in the graphical version of the F-Net. The fact that the graphical form does not express the mathematical transformation itself leaves it free of much of the complexity to which Brooks refers in his refutation of the possibility of workable, graphical languages [9].

5.7. Imperative Sequential Programs

An imperative sequential unstructured program can be converted to an F-Net by making each statement into an instruction, each variable into an m-variable having a single-element control domain, and the instruction counter into a variable with no data state but a very large control domain, with each possible control state representing the location of a statement. Each instruction would be bound to the appropriate data variables and to the proper control state of the instruction counter. The resultant F-Net would contain no concurrency aside from some possible optimistic buffering due to non-volatile arguments, and the graphical representation would resemble a rat's nest of arcs to m-variables, and a too-detailed depiction of control by the fingers within the instruction counter variable.

Instead of this fine-grained approach, an instruction in an F-Net typically consists of a group of logically-related statements, and m-variables contain logically-related groups of variables. The control state is moved away from the central program counter, instead being distributed among the "states of completion" (control states) of the program's variables (m-variables), enforcing only the order in which they are accessed by different instructions.

In this form, each instruction resembles a block in a structured language, and transitions are transfers among blocks. This transfer of control is looser than that employed in structured-programming practices, but the graphical F-Net provides a flowchart for these branches, uncluttered with the statement-by-statement control and data management that is better shown in the text of the program. Unlike traditional process models where control structures hide within the communicating processes, control state puts it between processes (instructions) so that decisions affecting subsequent process execution are made explicit in the model.

5.8. Conclusion

The comparisons here have regarded control state in a variety of different ways: as a predicate for a guard, as a means of ensuring atomic execution, as part of the name of a variable, and as a program counter. In addition, the elements of a variable's control domain can be considered the states of a finite state machine, which watches over access to the variable. We believe these examples illustrate the power present in the simple concept of control state.

CHAPTER 6

Implementation

6.1. Introduction

The axioms defining the semantics of F-Nets in Chapter 4 provide a basis for determining whether a specific implementation of the model is correct—i.e. whether the execution graphs produced by an F-Net in the implementation obey the axioms. However, the axioms were chosen with the additional goal of facilitating efficient implementations on a variety of architectures. This chapter will prove (by example) that this is indeed possible. First, a generic implementation will be developed and shown to obey the axioms. Then, this implementation will be optimized separately for shared-memory and message-passing architectures.

6.2. Definition of a Valid Implementation

Let F be the set of all legal F-Nets (minus isomorphisms), and X be the set of all legal execution graphs (minus isomorphisms). Then the semantics of F-Nets given in Chapter 4 can be considered as a function

$$\rho{:}F \rightarrow \mathbf{P}(X)$$

which maps each F-Net to the set of all legal execution graphs for that F-Net. The F-Nets and execution graphs described in Chapter 4 will be called abstract, and ρ will be called the interpretation function.

A specific host environment for F-Nets, which includes language processors, a run-time environment, and the host computer's architecture, can also be considered as the definition

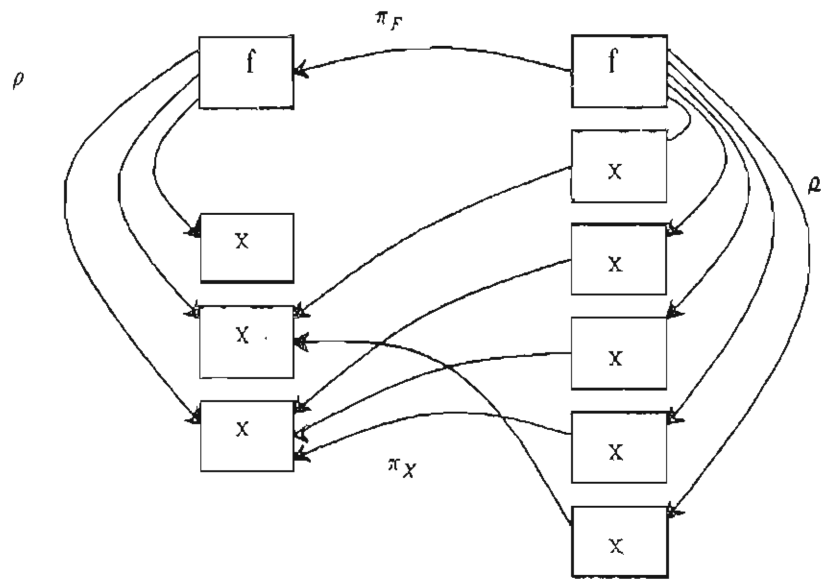


Figure 6.1 Relationship of Mappings for a Hypothetical F-Net

implementation to produce its concrete execution graph in a finite amount of time. In light of the fairness rule discussed in chapter 3, we instead require that any finite prefix subgraph of the execution graph be computed in a finite time. (We specifically do *not* require that this amount of time be predictable, nor even that it be possible to tell when the time has elapsed.) This means that an implementation must enlarge the graph evenly in some sense.

6.3. A Generic Implementation

In this section, a generic implementation will be proposed and argued to be valid. The first two subsections will define the forms for the concrete F-Net and concrete execution graphs in the generic implementation, and will describe the π_F and π_X mappings which interpret these as abstract F-Nets and execution graphs. The third subsection will concentrate on defining the ρ function which executes the concrete F-Net on an actual architecture to produce the concrete execution graph.

6.3.1. Concrete F-Nets (π_F)

The concrete C-based textual syntax that was presented informally in the Chapter 3 "sum of squares" example will be used for the generic implementation. A more formal description of the syntax follows:

fnet := `Vars` *var*$^+$ `Ops` *op*$^+$ `Instrs` *instr*$^+$
var := *type varname* (*cstatename*$^+$)
op := *opname* [*arg*$^+$] *opbody*
arg := *rwperms argname* (*transname*$^+$)
rwperms := `in` *type* | `out` *type* | `inout` *type* | `nodata`
instr := *opname* [*binding*$^+$]
binding := *argname* : *cstatename varname* (*transbdg*$^+$)
transbdg := *transname* : *cstatename*

where

varname, *cstatename*, *opname*, and *transname*

are legal C identifiers

type is a legal C type expression (possibly a structured type)

opbody

is a legal C program block with the following caveats:

(1) Statements which could facilitate communication with other programs are disallowed. This includes I/O. (Chapter 7 will address how I/O can be built into the model.)

(2) Arguments declared in the *arg* section which have *rwperms* other than `nodata` can be accessed within the program as though they were variables with the associated type, except that they cannot be aliased. `in` arguments cannot be used in any context in which their contents could be modified.

(3) Transition statements, of the form

< *transname argname* >

are included, where *argname* is an argument of the operation and *transname* is a legal transition of that argument, as identified in the operation signature.

6.3.2. Abstraction of Concrete F-Nets (π_F)

The concrete F-Net is abstracted in a fairly obvious way.[2] The declarations in L represent the following mappings:

Variable Declarations

The *var* productions represent a mapping:

$Var\_dec : Varname \rightarrow (CtlDom \rightarrow \mathbb{N})$

i.e. each variable name is associated with a mapping from the declared control domain of that variable to the natural numbers. The $CtlDom \rightarrow \mathbb{N}$ mapping is sequential within the variable (i.e. the first-mentioned control state to 1, the second to 2, etc.).

Operation Declarations

The *op* productions represent a mapping:

$Op\_dec : Opname \rightarrow (Argname \rightarrow (\mathbb{N} \times RWperm \times (Transname \rightarrow \mathbb{N})) \times Opbody)$

i.e. each operation name is associated with an argname mapping (corresponding to the *arg* productions), and an operation body. The argname mapping associates each argument name with (1) an element of $\mathbb{N}$, (2) a read-write permission (a member of {**in**, **out**, **inout**, **nodata**}), and (3) a mapping which associates each transition name with an element of $\mathbb{N}$. The $Argname \rightarrow \mathbb{N}$ mapping is sequential within the operation, the $Transname \rightarrow \mathbb{N}$ mapping is sequential within the argument.

Instruction Declarations

The *instr* productions represent a mapping

[2]If a formalized view of this "obvious" interpretation is not helpful, the reader can safely skip all but the discussion of ϕ in the last few paragraphs of this subsection.

$Instr\_dec{:}Instr \rightarrow$

$(Opname \times (Argname \rightarrow CtlDom \times Varname \times (Transname \rightarrow CtlDom)))$

Based on these mappings, the F-Net $\pi_F(\mathcal{L}) = \langle \Sigma, S, O, I \rangle$ is defined. Since the abstract model does not address the issue of types, Σ will be taken to be the set of strings of bits, and types will be mapped onto this set as appropriate. S will be the domain of Var_dec (i.e. the set of all variable names). O and I will be the range of the mappings $Ops{:}Opname \rightarrow O$ and $Instrs{:}Instr \rightarrow I$ which are defined below. (In these definitions, lower case names are free variables.)

$Ops{:}Opname \rightarrow O$

$Ops(opn)$ is defined as $\langle a, R, W, \phi \rangle$ where $Op\_dec(opn) = \langle sig, opbody \rangle$ and

$a \equiv |Domain(sig)|$

$R \equiv \{arg \mid sig(argn) = \langle arg, rwp, transs \rangle \wedge (rwp = \mathbf{in} \vee rwp = \mathbf{inout})\}$

$W \equiv \{arg \mid sig(argn) = \langle arg, rwp, transs \rangle \wedge (rwp = \mathbf{out} \vee rwp = \mathbf{inout})\}$

ϕ is defined via the behavior of the $opbody$ program, as described shortly.

$Instrs{:}Instr \rightarrow I$

$Instrs(instr)$ is defined as $\langle o, \beta, \gamma, \delta \rangle$ where $Instr\_dec(instr) = \langle opn, args \rangle$ and $Op\_dec(opn) = \langle sig, opbody \rangle$ and

$o \equiv Ops(opn)$

$\beta \equiv \{(arg, sw) \mid sig(argn) = \langle arg, rwp, transs \rangle$

$\wedge args(argn) = \langle pbnd, sw, tbnd \rangle\}$

$\gamma \equiv \{(arg, cstate) \mid sig(argn) = \langle arg, rwp, transs \rangle$

$\wedge args(argn) = \langle pbnd, sw, tbnd \rangle \wedge Var\_dec(sw)(pbnd) = cstate\}$

$\delta \equiv \{(arg, (trans, cstate)) \mid sig(argn) = \langle arg, rwp, transs \rangle \wedge transs(transn) = trans$

$\wedge args(argn) = \langle pbnd, sw, tbnd \rangle \wedge Var\_dec(sw)(cstate) = cstate \wedge tbnd(transn) = cstate\}$

In the definition of *Ops*, the result of $\phi(arg_1, arg_2, \cdots, arg_{|R|})$ is defined by the *opbody* program's behavior when it is executed after initializing (a) the argument variables corresponding to the opbody's `in` and `inout` arguments with the values $arg_1 \cdots arg_{|R|}$, and (b) the argument variables corresponding to the operation's `out` arguments with predetermined constant initialization values (e.g. zeroes). A transition statement has no effect on the execution except that any further reference to that argument, either within a transition statement or as a reference to the data values associated with it, will cause the program to halt. The result of the firing function,

$\langle res_1, res_2, \cdots res_{|W|}, t_1, t_2, \cdots t_a \rangle$

is the following interpretation of that execution:

t_{arg}:

If a transition statement executes for argument *arg*, then $t_{arg} \equiv trans$, where *trans* is the transition field from the first such statement executed. If *arg* is non-volatile, $t_{arg} \equiv 1$. If neither of the above is true, $t_{arg} \equiv \perp$.

res_k where *arg* is the *k*th write (`out` or `inout`) argument:

If a transition statement executes for argument *arg*, then res_k is the value assigned to the argument variables associated with *arg* when the first such statement executed. If a transition statement does not execute for *arg*, $res_k \equiv \perp$.

Although determining the result of evaluating ϕ with some arguments may be undecidable in some cases, it is nonetheless well-defined and meets the requirements for a firing function for an operation with that signature. Note that the initialization of `out` arguments to a constant is necessary; if they were not initialized, and the argument variables (or portions thereof) were not assigned new values during execution of the opbody program, then the remaining data values when a transition was performed would not necessarily be a function of the values present on the readable arguments when the execution began. This initialization also allows the program to read the values associated with an **out** argument without disturbing the functional nature of the required mapping, since the values read will either be the original constant values or new values which must already be a function of the values originally on the read arguments.

6.3.3. Concrete Execution Graphs and Their Abstraction (π_X)

A concrete execution graph will be of the form

$$x = \langle C,E,B,A,\hat{\sigma},\bar{\sigma},\epsilon,fair \rangle$$

where

$C,E,B,A,\hat{\sigma}$, and $\bar{\sigma}$ are identical in form to an abstract execution graph

$\epsilon{:}E \rightarrow Cont \times Store$ is the *Execution State* of e

$fair$ is the *Fairness State*

ϵ and $fair$ will be described in more detail later. Their role is to help determine the next action to perform on the concrete execution graph. The abstract interpretation of a concrete execution graph will be obtained by omitting ϵ and $fair$—i.e.

$$\pi_X(\langle C,E,B,A,\hat{\sigma},\bar{\sigma},\epsilon,fair \rangle) \equiv \langle C,E,B,A,\hat{\sigma},\bar{\sigma} \rangle$$

6.3.4. Concrete Implementation (ρ)

This section will describe a concrete implementation ρ. First, a strategy will be presented, then this strategy will be shown to meet the requirements for ρ: i.e. any finite prefix subgraph of the resulting concrete execution graph will obey the semantic axioms within a finite amount of time. Then, an implementation based on that strategy will be presented and shown to correctly implement the strategy.

6.3.4.1. Strategy

The strategy of the concrete implementation will be to define a mapping $\underline{\rho}':X \rightarrow X$ which takes a concrete execution graph and returns one which is somehow more well defined: i.e. is closer to obeying the semantic axioms than its predecessor. A partial ordering on execution graphs will be defined to formalize exactly what this means. $\underline{\rho}$ is then defined as the result of calling $\underline{\rho}'$ recursively on the empty execution graph, $\perp_X$ (i.e. $\underline{\rho}'(\underline{\rho}'(\cdots \underline{\rho}'(\perp_X) \cdots)))$ an infinite number of times.

To avoid referring back to concrete syntax, the abstract F-Net corresponding to the concrete F-Net (obtained by π_F described above) will be used as a precise notation for describing the implementation. The abstract definition of the firing function, ϕ, is not useful for implementation, however, since it was not defined in terms of the syntax of the *opbody* program, but its behavior.

To address this, we describe here, for each *opbody*, a continuation which maps an initial program store to a final program store, by treating the *opbody* as a C program, but augmenting the traditional store used by C, which we will call *cstore*, with four other kinds of store which will be acted upon by transitions and data state references. The overall store operated on by the *opbody* will be a tuple $store \equiv \langle cstore, tstore, fstore, sstore, estore \rangle$.

tstore: A set containing the arguments for which transition statements have been performed by the current execution.

fstore: A set containing the non-volatile arguments for which a transition has been automatically performed (by the Finishing step, defined below).

sstore: A vector of segments, indexed by argument, which contains the values for the operation's argument variables.

estore: A vector of $\mathbb{N}_\perp$ indexed by argument. which contains the transition performed to each of the operation's arguments.

The argument references in the *opbody* are now converted to executable code as follows. Each reference to an argument variable for argument *arg* causes `check_arg(`*arg*`)` to be executed before the appropriate access to *sstore*[*arg*] is performed, where

```
check_arg($arg$) ≡
    If $arg \in tstore$
        halt
```

Each transition statement <*trans arg*> in the opbody becomes `perf_trans(`*trans*, *arg*`)` where

```
perf_trans($trans$, $arg$) ≡
    check_arg($arg$)
    $tstore \leftarrow tstore \bigcup \{arg\}$
    if $arg \notin fstore$
        $estore[arg] \leftarrow trans$
```

With these translations, the *opbody* defines a continuation which maps an initial *store* to a final *store*. $\hat{\imath}\cdot o\cdot\phi'$ will refer to that continuation.

The $\dot{\sigma}$ and $\ddot{\sigma}$ mappings are defined for concrete execution graphs as follows: (Elements of *store* should be prefixed by $\epsilon(e)\cdot store\cdot$)

$$\dot{c}\equiv\begin{cases} 1 & \text{if } {}^*c=\{\} \\ \hat{e}\cdot\delta(arg)(estore[arg]) & \text{if } {}^*c=\{e\}\ (where\ arg=\hat{e}\cdot\beta^{-1}(\bar{c})) \end{cases}$$

$$\ddot{c}\equiv\begin{cases} \iota(\bar{c}) & \text{if } {}^*c=\{\} \\ sstore[arg] & \text{if } {}^*c=\{e\}\wedge arg\in(tstore\bigcup fstore)\setminus\hat{e}\cdot o\cdot W\ (where\ arg=\hat{e}\cdot\beta^{-1}(\bar{c})) \\ \perp & otherwise \end{cases}$$

Define $\rho'(x)\equiv x'$, where x' is identical to x except that one or more of the alterations described in the following four steps have been applied:

Initialization: Create a new initial c node

If $\exists v \in V\ \forall c \in C. \tilde{c} \neq s$ then create a new c node and define $\bar{c} \equiv s$.

Extension: Create new e node and successor c nodes

If $\exists i \in I\ \forall arg \in [1, i \cdot o \cdot a] \exists c \in C. c^* = \{\} \wedge \vec{c} = i \cdot \beta(arg) \wedge \dot{c} = i \cdot \gamma(arg)$ then

(a) Create a new e node, make it the successor for all the c nodes instantiated in the condition, define $\acute{e} \equiv i$, and define $\epsilon(e) \equiv \langle i \cdot o \cdot \phi', \langle cstore, \{\}, \{\}, sstore, estore \rangle\rangle$, where $cstore$ is the initial store as defined by the C implementation, all elements of $estore$ are $\bot$ and

$$sstore[arg] \equiv \begin{cases} 0 & \text{if } arg \in i \cdot o \cdot W \setminus i \cdot o \cdot R \\ \ddot{c} & otherwise\ (where\ c^* = e \wedge \vec{c} = i \cdot \beta(arg)) \end{cases}$$

(b) For all $arg \in [1, i \cdot o \cdot a]$, create a c node, make it a successor to e, and define $\bar{c} \equiv i \cdot \beta(arg)$

Execution: Advance the execution state of an e node

If $\exists e \in E. \epsilon(e) = \langle cont, store \rangle \wedge cont \neq \mathbf{halt}$,

then evaluate $cont(store)$ for a finite amount of time, yielding a new continuation $cont'$ and store $store'$. Define $\epsilon(e) \equiv \langle cont', store' \rangle$.

Finishing: Perform transition to a non-volatile argument

If $\exists e \in E,\ arg \in [1, \acute{e} \cdot o \cdot a]. arg \notin \acute{e} \cdot o \cdot W \wedge \acute{e} \cdot o \cdot \tau(arg) = 1$

$\wedge \epsilon(e) = \langle cont, \langle cstore, tstore, fstore, sstore, estore \rangle\rangle \wedge arg \notin tstore \bigcup fstore$

then define $\epsilon(e) \equiv \langle cont, \langle cstore, tstore, fstore', sstore, estore' \rangle\rangle$ where $fstore' = fstore \bigcup \{arg\}$ and $estore' = estore$ except that $estore'[arg] = 1$

The frequency in which these steps will be executed in subsequent iterations of ρ' will be constrained by the following fairness criterion:

For some set of free variable instantiations, the condition for a step will not remain true for more than a finite number of applications of ρ' before the step is executed.

The role of *fair* is to record enough history of which steps have been applied to guarantee the fairness constraint. For example, *fair* could be implemented as a simple vector of the number of applications of ρ' that have occurred that have not executed a particular step.

6.3.4.2. Validity of Implementation Strategy

Define the following partial order over concrete execution graphs X:

$$x \sqsubseteq x' \equiv (\forall c \in x \cdot C, c' \in x' \cdot C . c \cong c' \Rightarrow \dot{c} \sqsubseteq \dot{c}' \land \ddot{c} \sqsubseteq \ddot{c}') \land x \text{ is a prefix subgraph of } x'$$

i.e. execution graph x is less well defined than x' if x is a prefix subgraph of x' and the control and data states of all of the c nodes in x are less well defined than those in x'. The bottom element of this partial order is the empty execution graph.

By this definition, $x \sqsubseteq \rho'(x)$. That the x is a prefix subgraph of $\rho'(x)$ follows directly from the Initialization and Extension steps. That the new graph has more well defined $\dot{\sigma}$ and $\ddot{\sigma}$ functions is shown as follows. If a c node is introduced without predecessors (in the Introduction step), $\dot{c}$ and $\ddot{c}$ are well-defined, and since no step gives predecessors to a previously-existing c node, remains well defined. If a c node is introduced with a predecessor e (in the Extension step), $\dot{c}$ will have a non-$\bot$ value precisely when the argument of $\dot{e}$ bound to $\bar{c}$ is a member of $tstore \bigcup fstore$, and this condition cannot be made false nor can the value $estore[arg]$ be redefined once this condition is true. $\ddot{c}$ is defined only under the same condition or when $arg \notin \dot{e} \cdot o \cdot W$. In the former case, the value $sstore[arg]$ cannot be redefined for the same reasons as $\dot{c}$, and it cannot be redefined in the latter case due to the syntactic restriction that such argument variables cannot be used within the *opbody* in a context where their contents can be altered.

Thus, ρ' is monotonic in the sense that it does not move down or across the partial order. The remainder of this section shows that it not only moves up the partial order, but that the semantic axioms will hold for any finite prefix subgraph of the concrete execution graph within a finite number of recursive applications of ρ'. (Clearly, each application takes a finite time.)

Axiom 1: The property that

$$\forall s \in S \exists c_0 \in C. \bar{c}_0 = s \wedge \dot{c}_0 = 1 \wedge \ddot{c}_0 = \iota(s)$$

is ensured by fairness of Initialization and the definition of $\dot{\sigma}$ and $\ddot{\sigma}$. That these c_0s are members of C_0 (i.e. precede all members of C having the same name) follows from the fact that all other $c' \in C. \bar{c} = s$ are created in the Extension step, and each such node can be inductively shown to be preceded by a c node with the same name created in the Initialization step.

Axiom 2: A c node is given a successor only in the Extension step and only under the condition that it does not already have successors, and is given a predecessor only in the Extension step, and only under the condition that it is newly created (i.e. has no predecessors).

Axiom 3-Structure and Axiom 3-Condition: Only the Extension step creates e nodes or gives them predecessors or successors, and it obeys these axioms by definition.

Axiom 3-Result: The fairness constraint ensures that the Execution step will be repeatedly applied to every execution state until it halts. The continuous semantics of sequential computer languages ensures that if the sequential computation denoted by the concrete operation includes `perf_trans`, then it will be executed in a finite number of applications of ρ'. The fairness constraint also ensures that the Finishing step will be executed for all non-volatile arguments for which `perf_trans` has not executed within a finite number of calls. Axiom 3-Result follows directly from these and the definition of the $\dot{\sigma}$ and $\ddot{\sigma}$ labelings.

Axiom 4-Non-interference: Ensured by the initialization of *sstore* in the Extension step, and by the definition of $\ddot{\sigma}$.

Axiom 5-Liveness: Ensured by the fairness constraint applied to the Extension step.

Axiom 6-Time: No step gives a predecessor to an already existing node, and no step creates a cycle.

6.3.4.3. Pseudo-code for the Generic Implementation

This section presents an efficient implementation of the above strategy. First, it will be assumed that S is finite, so all applications of ρ' which perform the Initialization step can be performed first. Second, the Extension condition can be checked, and the Extension step executed, immediately whenever the control state of a "terminal" c node (i.e. one without successors) becomes defined within the Initialization or Finishing steps. The Execution step can be made efficient by keeping all $\langle$*cont*,*store*$\rangle$ pairs on a "run" queue, which is serviced in a fair, round-robin fashion. Entries need not remain on the queue for instructions which have halted or performed transitions to all of their arguments, since further advancement of execution will not affect any aspect of the execution graph other than the continuation or store itself. The precise time and method of invoking the Finishing step will not be detailed until later sections on implementations for specific architectures.

To facilitate efficient checking of the Extension condition and performance of the Execution step, the $\dot{\sigma}$ and $\ddot{\sigma}$ labelings will be maintained explicitly, and only for the terminal c nodes. The data state for the terminal c node for which $\bar{c}=s$ will be called *dstate*[s]. The control of these terminal c nodes state will not be maintained directly, but will be reflected in an vector called *reasons*, indexed by instruction: *reasons*[i] will equal n if the c nodes for n of its argument bindings either do not exist or the terminal nodes have a control state other than that of the instruction's firing constraint. Thus, the condition in the Extension step will reduce to testing whether the *reasons* count for that instruction is 0.

With these changes, *estore* becomes redundant—all assignments $estore[arg] \leftarrow trans$ will be replaced with calls to dec_count$(i \cdot \beta(arg), i \cdot \delta(arg)(trans))$, which will decrement the reasons for all instructions bound to the variable and constrained by the control state mentioned in the arguments, and check whether the execution condition has been met for those instructions. This requires that perf_trans has access to i (the instruction on whose behalf the operation is executing), so we augment the store with *istore* which contains the current instruction. The *store* now consists of $\langle cstore, istore, tstore, fstore, sstore \rangle$

Portions of the execution graph which will no longer be used in performing the steps will simply be discarded. All side-effects of graph creation will be present even though the graph itself will not be maintained. Annotations will relate each portion of the code to the steps above so that code to keep the graph can be added if desired, say for debugging purposes.

perf_trans (*trans*, *arg*) now becomes

```
perf_trans($trans$, $arg$) $\equiv$
    check_arg($arg$)
    $tstore \leftarrow tstore \bigcup \{arg\}$

***Define $\dot{c}=istore \cdot \delta(arg)(trans)$ where $c$
***         is the oldest node such that $\bar{c}=istore \cdot \beta(arg)$
***If $arg$ has write usage, define $\dot{c}=sstore[arg]$

    if $arg \in istore \cdot o \cdot W$
        $dstate[istore \cdot \beta(arg)] \leftarrow sstore[arg]$
    if $arg \notin fstore$
        dec_count($istore \cdot \beta(arg)$, $istore \cdot \delta(arg)(trans)$)
    if $tstore \bigcup fstore=[1,istore \cdot o \cdot a]$
        halt
```

Since we assume that *fstore* is lost when an evaluation finishes, the "normal halt" (executed after the last line of code in an operation) must now ensure that all non-volatile arguments are finished:

```
normal_halt ≡
    for $arg \in [1, istore \cdot o \cdot a]. arg \notin istore \cdot o \cdot W \wedge istore \cdot \tau(arg) = 1$
      if $arg \notin fstore$
         fin_trans($arg$)
```

The concrete implementation (minus invocation of the finishing step) is `conc_imp`, where:

```
conc_imp ≡
    init0
    init1 | serve_q | serve_q | serve_q | ···

init0 ≡

***Initialize the execution graph to be empty

    for $i \in I$
        $reasons[i] \leftarrow i \cdot o \cdot a$

init1 ≡
    for $s \in S$

***    INITIALIZATION:  Create $c$ node, define $\bar{c} \equiv s$, $\dot{c} \equiv 1$, $\ddot{c} \equiv \iota(s)$

        $dstate[s] \leftarrow \iota(s)$
        dec_count ($s$, 1)

serve_q ≡
    repeat
        $\langle cont, store \rangle \leftarrow$ dequeue

***    EXECUTION

        $\langle cont', store' \rangle \leftarrow$ tslice ($cont, store$)
        if $cont' \neq$ halt
           enqueue ($\langle cont', store' \rangle$)
    forever

dec_count ($s$, $n$) ≡

***For each instruction which benefits from new control state
***      of $c$ node...
```

```
a0: for $i \in I.\ \exists arg \in [1, i\cdot o\cdot a].\ i\cdot\beta(arg)=s \wedge i\cdot\gamma(arg)=n$

***   Decrement number of reasons it cannot fire

 a1:  $reasons[i] \leftarrow reasons[i]-1$

***   If no more reasons (i.e. Extension condition true) ...

 a2:  if $reasons[i]=0$

***       EXTENSION:
***       Create new successor $c$ nodes with $\perp$ control state

 a3:      inc_counts($i$)

***       Create new $e$ node, define $\dot{e} \equiv i$, initiate associated
***           opbody

          initiate($i$)

inc_counts ($i$) $\equiv$

***For each argument of new $e$ node...

    for $arg \in [1, i\cdot o\cdot a]$

***   Let $c'$ be oldest $c$ node such that $\bar{c}'=i\cdot\beta(arg)$
***   Create new $c$ node, define $\bar{c} \equiv i\cdot\beta(arg), \dot{c} \equiv \perp$
***   If $arg$ has no write usage, define $\dot{c} \equiv \dot{c}'$
***   For each instruction which benefited from control state
***         of $c'$ ...

 a4:  for $i2 \in I.\ \exists arg2 \in [1, i2\cdot o\cdot a].\ i2\cdot\beta(arg2)=i\cdot\beta(arg) \wedge i2\cdot\gamma(arg2)=i\cdot\gamma(arg)$

***       Increment number of reasons it cannot fire

          $reasons[i2] \leftarrow reasons[i2]+1$

initiate ($i$) $\equiv$
    for $arg \in i\cdot o\cdot R$
        $sstore[arg] \leftarrow dstate[i\cdot\beta(arg)]$
    for $arg \in i\cdot o\cdot W \setminus i\cdot o\cdot R$
        $sstore[arg] \leftarrow 0$
    enqueue $(\langle i\cdot o\cdot\phi', \langle cstore, i, \{\}, \{\}, sstore\rangle\rangle)$
```

The Finishing step will consist of performing `fin_trans`(arg) for non-volatile argument arg, where

```
fin_trans($arg$) ≡
    if $arg \not\subseteq tstore$
        $fstore \leftarrow fstore \bigcup \{arg\}$

***     FINISHING:  Define $\dot{c} \equiv istore \cdot \delta(arg)(1)$, where $c$
***             is oldest node such that $\bar{c} = istore \cdot \beta(arg)$

        dec_count  ($istore \cdot \beta(arg)$,  $istore \cdot \delta(arg)(1)$)
        if  $tstore \bigcup fstore \equiv \{1, istore \cdot o \cdot a\}$
            halt
```

The optimal time and method of invoking fin_trans (other than that in normal_halt) will depend upon the specific architecture.

In order to ensure that the Extension step executes only when the condition is true, the test at a2 and the execution of inc_counts at a3 must together occur atomically. In addition, the decrement at a1 must occur as an atomic step to avoid read-write conflicts. These atomicity constraints could be ensured by providing that only one instruction perform dec_counts at a time (perhaps by funneling all executions through a common monitor or acquiring a global lock) or by separately assigning each *reasons* count its own lock and locking all that could be accessed by inc_counts before a2, but the former solution creates a global bottleneck while the latter creates significant overhead in dealing with each instruction separately and avoiding deadlock. A middle ground can be reached by defining a *rivals* relation $\ll\gg$ as

$$i \ll\gg i' \equiv \exists s \in shared(i, i').\, i \cdot b(i \cdot \beta^{-1}(s)) = i' \cdot b(i' \cdot \beta^{-1}(s)))$$

(i.e. two instructions are rivals whenever their firing constraint uses the same control state at least one variable) and a rivalry as the transitive closure of $\ll\gg$. By this definition, all instructions referenced in the for at a0 plus all instructions referenced in the for at a4 will belong to the same rivalry. We will enforce the atomicity constraints by restricting execution of dec_counts to one instruction per rivalry at a time. (Note that the call to initiate does not need to be protected, but moving it outside of the dec_counts logic requires additional bookkeeping overhead.)

6.4. Optimizing Concrete Execution for Differing Architectures

Both a shared-memory and message-passing implementation will be presented. Both will be based on the generic implementation described in the last section, but will differ in the details of copying and maintaining *dstate* and *sstore*, and deciding when to invoke `fin_trans`.

6.4.1. Shared Memory

The primary focus behind the shared memory implementation will be to reduce or eliminate copying of *dstate* to *sstore* in `initiate`. For the most part, this is accomplished by using the *dstate* directly in place of the *sstore* (i.e. replacing the $sstore[arg] \leftarrow dstate[s]$ statements with $sstore[arg] \equiv dstate[s]$).

The flaw with this simple implementation is that an instruction may continue to access *dstate* after `fin_trans` has been performed, and `fin_trans` may allow another instruction to begin execution and access the same *dstate* concurrently with the current instruction. If the subsequent instruction does not have write usage to the variable, or the current instruction does not have read usage, no harm is done—the memory can be shared by both instructions. If, however, an instruction with write usage to the variable is initiated while an instruction is still reading the variable, they clearly cannot access the same *dstate* buffer without disastrous consequences. In this case, a new version of the *dstate* for that variable must be allocated and used by the new instruction, leaving the old version to be deallocated when the last reader has finished with it. If the new instruction has read usage to the variable (in addition to its write usage), the contents of the old version must be copied to the new one.

The following replacements of `initiate` and `perf_trans` are based on the above descriptions. Instead of containing data state, *dstate*, and *sstate* (and m) now contain pointers to a tuple of $\langle readers, version, data\rangle$, where *readers* contains the number of instructions reading the data state, *version* is the version number of the data state, and *data* is the actual data state. An auxiliary vector, *version*, contains the number of the latest version for each variable. `alloc` allocates a data state tuple and returns a pointer to it, `dealloc` deallocates the data state associated with such a pointer. Indirection is represented by *.

```
initiate (i) ≡
    $s \equiv i \cdot \beta(arg)$
    for $arg \in i \cdot o \cdot W$
        if $^*dstate[s] \cdot readers \neq 0$
            $m \leftarrow$ alloc()
            if $arg \in i \cdot o \cdot R$
                $^*m \cdot data \leftarrow {}^*dstate[s] \cdot data$
            $dstate \leftarrow m$
            $^*dstate[s] \cdot readers \leftarrow 0$
            $version[s] \leftarrow version[s]+1$
            $^*dstate[s] \cdot version \leftarrow version[s]$
        if $arg \notin i \cdot o \cdot R$
            $^*dstate[s].data \leftarrow 0$
    for $arg \in i \cdot o \cdot W \bigcup i \cdot o \cdot R$
        $sstore[arg] \leftarrow dstate[s]$
    enqueue $(\langle i \cdot o \cdot \phi', \langle cstore, i, \{\}, \{\}, sstore\rangle\rangle)$

perf_trans($trans$, $arg$) ≡
    check_arg($arg$)
    $tstore \leftarrow tstore \bigcup \{arg\}$
    $s \equiv istore \cdot \beta(arg)$
    if $arg \in istore \cdot o \cdot R$
        $^*sstore[s] \cdot readers \leftarrow {}^*sstore[s] \cdot readers - 1$
        if $^*sstore[s] \cdot readers = 0 \wedge sstore[s] \cdot version \neq version[s]$
            dispose($sstore[s]$)
    if $arg \notin fstore$
        dec_count($s$, $istore \cdot \delta(arg)(trans)$)
    if $tstore \bigcup fstore = [1, istore \cdot o \cdot a]$
        halt
```

In addition, all other references to $sstore[arg]$ must be replaced by $^*sstore[arg] \cdot data$. To ensure that only one instruction per rivalry performs the `dec_count` code, each rivalry must be assigned a lock which `dec_count` must acquire to execute.

There are other ways to optimize this implementation. The above tricks with pointers (and thus the extra indirection) can be avoided completely for variables with no non-volatile readers, or where it can be statically determined that a non-volatile reader will never relinquish the variable to a writer. Also, `fin_trans` can be delayed for any finite amount of time, and doing so may allow a reader to finish before a writer to the same variable is initiated, avoiding the need for making a copy. In fact, each time another reader of the variable is initiated, this time starts over, so if reader initiations progress continuously, a writing instruction may be postponed indefinitely without violating the fairness in the model. In the general case, however, parallelism can be maximized by performing `fin_trans` as soon as possible after the e node is created. The conflict between parallelism and copying overhead can be addressed directly by allowing the F-Net programmer to include hints within the concrete F-Net code.

6.4.2. Message-Passing

The message-passing based implementation described here will assume that the instructions are statically assigned to processors. This assignment can be performed randomly, or based on a static analysis of the F-Net together with a knowledge of the interconnect network present in the hardware.

The functionality of the implementation is distributed among processes, both by replicating some of the procedures (such as `dec_count`) and by splitting single procedures (such as `init0` and `initiate`).

Initialization Process

An initialization process will contain the code for the main process:

```
init_process ≡
   for $i \in I$
      spawn instruct_process[$i$]
   for $r \in$ set of rivalries
      spawn rivalry_process[$r$]
   init1
```

The code for init0 is distributed among the rivalry processes.

Rivalry Processes (One per Rivalry)

A rivalry process will perform the init0 code for its rivalry, then will await messages requesting that dec_count be invoked for an instruction in the rivalry. The process will contain the code for dec_count and the procedures called by it (inc_counts and initiate) and will locally maintain all *reasons* counts associated with the rivalry, as well as all *dstate* entries during the time that they might be needed by an instruction in the rivalry (i.e. while the control state of that variable corresponds to a control state belonging to the rivalry).

```
rivalry_process [$r$] ≡
   for $i \in$ rivalry $r$
      $reasons[i] \leftarrow i \cdot o \cdot a$
   repeat
      await message $\langle i, arg, trans, newdstate \rangle$
      $s \equiv i \cdot o \cdot \beta(arg)$
      $n \equiv i \cdot o \cdot \delta(arg)(trans)$
      $dstate[s] \equiv newdstate$
      dec_count ($s$ , $n$)
   forever
```

initiate will be replaced by

```
initiate ($i$) ≡
   for $arg \in |i \cdot o \cdot a| \setminus i \cdot o \cdot W$
      $sstore[arg] \equiv dstate[i \cdot \beta(arg)]$
   send $\langle sstore \rangle$ to instruct_process[$i$]
```

The missing *sstore* initialization code is performed within the instruction process. Note that even the *sstore* for arguments which are not read nor written is sent to the instruction process, since the *sstore* must be forwarded to the new rivalry when a transition is performed.

Instruction Processes (One per Instruction)

An instruction process contains the code corresponding to the opbody corresponding to the instruction's operation.

```
instruct_process [$i$] ≡
   repeat
      await message $\langle sstore \rangle$
      for $arg \in i \cdot o \cdot W \backslash i \cdot o \cdot R$
         $sstore[arg] \leftarrow 0$
      $tstore \leftarrow \{\}$
      $fstore \leftarrow \{\}$
      for $arg \in [1, i \cdot o \cdot a].arg$ is non-volatile
         fin_trans ($arg$)
      enqueue ($i \cdot o \cdot \phi'$, $\langle cstore, i, \{\}, \{\}, sstore \rangle$)
   forever
```

The calls to dec_count(s, n) within perf_trans and fin_trans are replaced by

send $\langle istore, arg, trans, sstore[arg] \rangle$ to rivalry_process [r]

where r is the rivalry corresponding to the new control state. The if $\cdots$ $dstate[s] \equiv \cdots$ code should also be removed from perf_trans, since the instruction process no longer maintains *dstate*.

In cases where a rivalry contains exactly one instruction, the functionality of the rivalry process can be combined with that of the instruction process. Further optimizations based on the F-Net topology and read-write usages of arguments can minimize the number of times a *dstate* must be passed by bypassing the rivalry process in some circumstances. Shared memory techniques can also be used to avoid messages when multiple instructions reside on the same processor.

6.4.3. Final Implementation Notes

It has been noted that parallelism is increased by performing the `fin_trans` operation as soon as possible after initiation of the instruction. Taking this to its natural extreme, `fin_trans` can be called as part of the `initiate` code so that the results can be felt even before the execution of the opbody has begun. If this is done, a safeguard must be taken to ensure that looping does not occur within `initiate` when a *useless* subgraph is entered. This is a subgraph of an F-Net where the semantics dictate that each instruction in the sub-graph can fire again if it fires once, due to all of the instruction's arguments being non-volatile and always allowing another instruction (perhaps itself) with the same characteristics to fire. To prevent this, useless subgraphs can be detected syntactically before execution and handled specially.

The data flow nature of the model suggests an alternate implementation for message-passing architectures. In it, the instruction processes do not pass data state to the rivalry process on each transition, but instead report the location (processor and address) of the data state. The rivalry process now has three jobs: (a) to determine when an instruction can fire (as before), (b) to determine the optimal processor on which to execute that instruction (based on the locations of all of the variables and the required *opbody* program, as well as the load on each processor) and (c) to direct message traffic to move the code and data to the optimal processor. Such an implementation would benefit from an architecture in which one processor (i.e. the rivalry's) could initiate messages from a second processor (i.e. that containing the data state or opbody) to a third processor (i.e. the optimal processor). Under some circumstances, data state and/or opbody code could be preemptively sent to a probable optimal processor before the instruction is ready to fire, or even replicated onto several possibly-optimal processors. Unlike other process-migration techniques, no run-time state would need to accompany the opbody, and unlike standard dataflow techniques, data would not move to a "waiting-matching" store before moving to its final destination.

CHAPTER 7

Future Directions and Conclusions

7.1. Introduction

This work has presented F-Nets primarily as a theoretical model, although an implementation has been outlined to demonstrate its portability. This chapter will speculate on uses that might be made of the model, and on how it might be made more usable. The first section will propose some extensions to the model that could make it amenable to building large software systems. We then outline how F-Nets could provide leverage in developing software-engineering tools. Finally, we propose an architecture to efficiently execute F-Net programs.

7.2. Extensions to F-Nets

The F-Net model provides a framework for building operations which are atomic, deterministic, and stateless, and then for constructing a concurrent program from these operations. In practice, programs are not built or analyzed in this monolithic fashion: they consist of fragments, or modules, which are built separately and then composed. This ability to compose modules is a primary selling point of object-oriented programming, for example.

F-Nets can be modified to accommodate a very simple form of composition by augmenting F-Net fragments, or modules, with arguments similar to operations. This allows an F-Net instruction to bind either an operation or an F-Net module into another F-Net. The arguments are represented within the module implementation as formal m-variables, to which instructions within the implementation can be bound. This same extension can allow an F-net to bind to other "outside-world" objects, such as the input and output streams provided by an operating system. In itself, this extension does not make F-Nets more powerful, simply decomposable.

An extension which does add power is the treatment of modules and operations as first-class objects. In other words, an m-variable can contain the description of an operation or module as its data state, and a special `bind` operation is provided which takes such a description as an argument and "becomes" that operation or module when it fires. By allowing bindings to also be first class objects, also to be fed to the `bind` instruction, dynamic binding becomes possible.

With these extensions, the primary difference between F-Nets and object-oriented programming is persistence. This can be added by providing another operation, called `instantiate`, which takes a module description and returns another module description which is equivalent to the first except that the m-variables within are instantiated: i.e. whenever and wherever the module is bound, it will use the same copies of the m-variables with the same control and data states. An instantiated module can therefore be passed around as an object. When there is a need to access the object, it can be bound with a `bind` operation, allowing some of the instructions within to fire and access or alter the m-variables ("instance variables").

To provide a method of passing arguments through multiple levels of binding, arguments (arcs) can be extended to represent a bundle of arguments, called a *cable*. These cables can be nested hierarchically, allowing entire contexts of variables to be brought into out of, or through a level of module hierarchy. This allows for a very precise, controlled, yet flexible management of scope, foreign to most object-oriented languages.

A much more basic extension that must be provided to F-Nets before they are usable in an applications environment is the ability to partition m-variables. With the semantics outlined in this dissertation, if an entire array is present on a single m-variable, it is only accessible to one instruction at a time, regardless of whether other instructions would attempt to access the same elements. By creating arrays of m-variables and methods to bind arguments to portions of these arrays, this access contention can be addressed within the semantics of the model and handled properly by a scheduler, though perhaps at the cost of higher overhead per element.

7.3. Software Tools

Four inter-related features of F-Nets make them particularly suitable as a basis for tools:

(1) The parallel and sequential aspects of a program are specified separately, each in a form best suited to its function, so tools do not need to combine their approach to these very different aspects of syntax and execution.

(2) The uniform graphical representation for parallel aspects of the program, across tools and architectures, facilitates tool integration and similarity of user interface.

(3) The use of traditional sequential languages for implementing operations provides for the use or adaptation of existing sequential tools.

(4) The model seen by the user is very similar to the model used during execution, facilitating a "What you see is what you get" approach for execution-based tools.

7.3.1. Debugging/Monitoring Tools

Parallel debugging is regarded as a very difficult problem for several reasons. Sequential debugging techniques are not easily adapted to parallel programs, due to the lack of global program state or a single program counter. Non-deterministic execution makes re-creation of errors difficult, and this can be complicated even more when program execution timings are affected by the debugging process. Tracing the flow of control and data, or just determining what the correct flows should be, is difficult. If the program was created by a parallelizing compiler or tool, the user may not be familiar with the relationship between the source program and the program being debugged.

F-Nets address each of these problems. Because an F-Net consists of a graphical network, with each node representing a sequential program which exhibits sequential behavior, these aspects of the program can be approached separately. A high-level graphical interface [14] could be used to visualize and control the parallel behavior of the F-Net, and a traditional sequential debugger could be invoked for the low-level operation executions when needed. The desired flow of control and data is apparent in the static F-Net, and the actual flows could be easily represented by highlighting various portions of the graphical representation during an execution. If the software-engineering features of F-Nets lure users to create programs using the model, the execution behavior of the program during debugging will not be surprising or unfamiliar. Finally, the logging techniques described in Chapter 4 may well be non-intrusive enough so that every execution could be logged. If this is the case, any execution could be re-played within a debugger with exactly the same behavior as the original.

Perhaps the most important debugging feature provided by F-Nets is the likelihood that they would not need to be debugged at all. In traditional process models, the user strategically places synchronization and communication primitives in the program in order to obtain the desired execution behavior, while with F-Nets, the user explicitly specifies the desired execution behavior, decreasing the chances for accidental communication or synchronization.

7.3.2. Parallel Restructuring Tools

Parallel restructuring tools take a sequential program and produce a parallel program. Taking a simplistic view, the resulting program is to have identical input/output behavior as the original sequential program—i.e. it is to represent the same function from input history to output history, but optimized to run well on some specific parallel architecture.

This view is too simplistic. To hold to a rigid semantic mapping thwarts much of the possible optimization which could occur for almost any parallel or vector architecture. A simple example of a slight alteration in semantics is the order in which a floating point sum reduction is performed. Since round-off errors will occur in different ways depending upon the order in which elements are summed, altering this order can change the behavior of the program.

The programmer has no way of specifying, in a purely sequential language, whether the order of the reduction is or is not important. If this information is conveyed to an interactive parallelizing tool, it is typically captured in a form which is unique to a particular architecture: restructuring the same program for another architecture would require the information to be specified again.

The problem above is a result of the fact that both the source and the result of restructuring or parallelizing is a specific implementation, rather than a description of the variety of implementations acceptable to the programmer. A more productive approach might be to view restructuring in two-steps:

(1) Converting a program into a more general version—i.e. providing non-deterministic choices for implementation of various constructs—such that the input program is one instance of the output program. This process will not be automatic, since the more general program can be regarded as the specific program plus human knowledge of

the specification. This generalization process could be iterative, with the more general program serving as input later to create an even more general program.

(2) Targeting a general program for a specific architecture. This could be done automatically or with user interaction. The resulting program should always match the specification of the general program, and may or may not be more specific (i.e. it may preserve the generality of the specification in the form of non-determinism, or it may "hard-wire" non-deterministic choices to some specific implementation).

For this approach to succeed, the programmer must be able to understand the output program from step 1 and to accept it as *the* program to replace the original. The F-Net model may be able to aid in this. The implementations described in Chapter 6 are examples of automatic targeting for parallel architectures.

7.3.3. Real-time Programming

F-Nets do not contain timing information only because this would clearly violate their architecture-independent qualities. Even on a given architecture, timing can change significantly depending upon the policy used by the scheduler. However, if these factors are known, the structure and simple semantics provided by F-Nets could help in addressing real-time programming problems. By approximating the time that each argument of each operation would take to produce a transition after the operation fired, then utilizing information about the architecture and scheduling policy to determine appropriate compositions for these timings, best- and worst-case scenarios could be computed to determine the effectiveness of the design or to alter scheduling decisions.

Non-determinism could also play an important role in real-time programming by serving as a means of specifying alternate actions in emergency (near-deadline) situations. This would require additional F-Net notation to specify when such an emergency action should occur, but it would not violate the F-Net semantics in any case: whether or not the emergency occurred, one of the non-deterministic actions would be taken, so the execution graph would still be valid.

7.4. Parallel Architecture

Although F-Nets have been proposed as an architecture-independent programming solution, specially developed architectures could take maximum advantage of the concurrency expressed in the model. The division between the traditional sequential nature of the operations and the control/communication nature of the rest of the F-net could be reflected in the architecture design, even without using special-purpose hardware, and the fact that F-Nets are designed to work in both high- and low-latency ether could allow for both low-latency shared-memory within clusters of processors and a high-latency scalable interconnect between clusters.

We propose an architecture consisting of clusters of processors, with each cluster consisting of one or more operation processors (OP) to execute operation programs, and a net processor (NP) to execute the net (i.e. perform scheduling decisions) and feed ready F-Net instructions to the OPs. Both the NP and OPs read and write a common Node Memory, with OP accesses mapped through an MMU. Two short queues are kept in Node Memory for fast communication between the NP and OPs: a queue of operations which are ready to execute on this cluster, written by the NP, and an event queue written by the OPs. The NPs in different clusters can communicate with each other via an interconnection network.

Since the OPs execute all user code, they should consist of very powerful processors. An OP repeatedly accesses the ready queue, obtaining pointers to an operation (i.e. a program implementing an operation) and to each the data states for its arguments, then initializes the MMU for these segments and begins executing the program. When the operation performs a transition, the corresponding segment is removed from the memory map, and a transition event is enqueued containing the argument and transition. If the program halts or a memory protection violation occurs, all segments are removed from the memory map, a halt event is enqueued, and the ready queue is read again.

The NP executes no user code, so it does not need excessive computation power or floating point capability. The NP serves four roles: (1) a "manager" for some set of rivalries in a net (2) a slave to the OP, (3) a slave to the interconnect (or more properly, to rivalry managers on other NPs), and (4) a stopwatch for the OP.

(1) As manager for some rivalries, it keeps the current reasons counts for all instructions within those rivalries, and keeps track of the locations of the program segment and data segments needed to execute those instructions. When an instruction's reasons count reaches zero, it is charged with the responsibility of determining the best node to execute the instruction on, based upon each node's current load and locality to these segments, and directing communication to unify these segments on that node. It may direct extra copies of data and program segments to several nodes to provide maximum flexibility in choosing a node to schedule an instruction.

(2) As slave to the OP, it monitors the event queue and relays the effects of events to the rivalry manager which is affected.

(3) As slave to the interconnect, it receives commands from other rivalry managers which direct it to send and receive program and data segments and to report on current load information. When a complete instruction (the program and all data segments) has been received, it enqueues this information in the ready queue for the OP.

(4) As stopwatch for the OP, it monitors the amount of time that the OP has been executing the current instruction. If there are more instructions in the ready queue, it interrupts the OP causing it to perform a context switch (i.e. go back to the ready queue). The assumption is that the OP will not need to timeslice between instructions in the general case.

More study would certainly need to be performed before pinning down the parameters for this architecture, but the Cogent XTM [31] offers one interesting possibility for an interconnect: both a high-speed bus for load status updates and data transfer commands, and a separate set of reconfigurable channels to handle the actual transfer of data and program segments between node processors.

Adding disks to each NP could make this architecture effective at handling database applications while also allowing little-used segments to be stored on disk rather than in node memory. Fault tolerance over single-OP or single-cluster failures could be implemented by having NPs create additional copies of data segments after each transition as well as keeping track of transitions which had not been relayed to other NPs.

7.5. Conclusion

F-Nets have been shown to be based on rational motivations: architecture independence, similarity of algorithm to computation, and the preservation of sequential semantics where possible. A construction has been proposed based on only these factors, and the result has been demonstrated to fulfill its goals. Architecture-independence has been shown through actual implementation techniques, in Chapter 6. Similarity of algorithm to computation has been shown formally, in Chapter 4. The preservation of sequential semantics of each operation implementation was built into the model during its construction in Chapter 3, and has been demonstrated in other chapters. In addition to achieving these goals, F-

Nets have been shown to be general enough to apply to a number of areas, providing a common ground on which to compare and contrast other formal models and establishing a basis for new tools and programming techniques.

References

[1] Agha, G. and Hewitt, C., "Concurrent Programming Using Actors," in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro (ed.), Cambridge, MA, MIT Press, 1987, pp. 37-53.

[2] Ahamad, M., Hutto, P. W. and John, R., "Implementing and Programming Causal Distributed Shared Memory," GIT-CC-90-49, College of Computing, Georgia Institute of Technology, 1990.

[3] Allen, J. R. and Kennedy, K., "A Parallel Programming Environment," *IEEE Software*, vol. 2, 4 (July 1985), pp. 21-29.

[4] Allen, R. and Kennedy, K., "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, vol. 9, 4 (October 1987), pp. 491-542.

[5] Athas, W. C. and Seitz, C. L., "Multicomputers: Message-Passing Concurrent Computers," *Computer*, vol. 21, 8 (August 1988), pp. 9-24.

[6] Babb, R. G. and DiNucci, D. C., "Design and implementation of parallel algorithms with Large-Grain Data Flow," in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon and R. J. Douglass (ed.), Cambridge, MA, MIT Press, 1987, pp. 335-349.

[7] Beguelin, A. L., "SCHEDULE: A Hypercube Implementation," *3rd Conference on Hypercube Concurrent Computers and Applications*, vol. I, Architecture, Software, Computer Systems and General Issues(January 1988), pp. 468-471.

[8] Boyle, J., Butler, R., Glickfeld, B., Disz, T., Lusk, E., Overbeek, R., Patterson, J. and Stevens, R., *Portable Programs for Parallel Processors*, New York, NY, Holt, Rinehart and Winston, 1987.

[9] Brooks, F. P., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, vol. 20, 4 (April 1987), pp. 10-19.

[10] Callahan, D. and Kennedy, K., "Compiling Programs for Distributed-Memory Multiprocessors," *Journal of Supercomputing*, vol. 2(1988), pp. 151-169.

[11] Carriero, N. and Gelernter, D., "Linda in Context," *Communications of the ACM*, vol. 32, 4 (April 1989), pp. 444-458.

[12] Chandy, K. M. and Misra, J., *Parallel Program Design: A Foundation*, Reading, MA, Addison-Wesley, 1988.

[13] DeMarco, T., *Structures Analysis and System Specification*, New York, NY, Yourdon Press, 1978.

[14] DiNucci, D. C., "Design of a debugger for large-grain dataflow programs," Technical Report CSE-88-005, Oregon Graduate Center, 1988.

[15] DiNucci, D. C. and Babb, R. G., "Design and implementation of parallel programs with LGDF2," *COMPCON'89*, San Francisco, 1989, pp. 102-107.

[16] Dongarra, J. J. and Sorensen, D. C., "A portable environment for developing parallel FORTRAN programs," *Parallel Computing*, vol. 5, 1&2 (July 1987), pp. 175-186.

[17] Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L., "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, 11 (November 1976), pp. 624-633.

[18] Foster, I. and Taylor, S., *Strand: New Concepts in Parallel Programming*, Englewood Cliffs, NJ, Prentice-Hall, 1989.

[19] Fuggetta, A., Ghezzi, C., Mandrioli, D. and Morzenti, A., "VLP: A Visual Language for Prototyping," *IEEE Workshop on Languages for Automation*, August 1988.

[20] Gopinath, K. and Hennessy, J. L., "Copy Elimination in Functional Languages," *Proceedings of the Conference on Programming Languages (ACM Symp. on Prin. of Programming Languages)*, 1989.

[21] Guarna, V. A., Gannon, D., Gaur, Y. and Jablonowski, D., "FAUST: An Environment for Programming Parallel Scientific Applications," *Proceedings Supercomputing '88*, Orlando, FL, November 1988, pp. 3-10.

[22] Hoare, C. A. R., *Communicating Sequential Processes*, Englewood Cliffs, NJ, Prentice-Hall, 1985.

[23] Jones, G. and Goldsmith, M., *Programming in occam 2*, Prentice-Hall, 1988.

[24] Jordan, H. F., Benten, M. S., Alaghband, G. and Jakob, R., "The Force: A Highly Portable Parallel Programming Language," *Proceedings of the 1989 International Conference on Parallel Processing*, vol. II - Software(August 1989), pp. 112-117.

[25] Kahn, G. and MacQueen, D. B., "Coroutines and Networks of Parallel Processes," *Proc. IFIP 77*, August 1977, pp. 993-998.

[26] Kaplan, I., "Programming the Loral LDF 100 Dataflow Machine," *ACM SIGPLAN Notices Notices*, vol. 22, 5 (May 1987), pp. 47-57.

[27] Karp, A. H. and Babb, R. G., "A comparison of 12 parallel fortran dialects," *IEEE Software*, 1988, pp. 52-67.

[28] Knuth, D. E., *The Art of Computer Programming: Volume 1/Fundamental Algorithms*, Reading, MA, Addison-Wesley, 1975.

[29] Li, K. and Hudak, P., "Memory Coherence in Shared Virtual Memory Systems," *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, Calgary, Alberta, Canada, August 1986, pp. 229-239.

[30] McGraw, J., Skedzielewski, S., Allan, S., Oldehoeft, R., Glauert, J., Kirkham, C., Noyce, B. and Thomas, R., "SISAL: Streams and Iteration in a Single Assignment Language: Language Reference Manual, Version 1.2," M-146, Rev. 1, Livermore, CA, Lawrence Livermore National Laboratory, March 1985.

[31] Merrow, T. and Henson, N., "System Design for Parallel Computing," *High Performance Systems*, January 1989, pp. 36-44.

[32] Milner, R., *A Calculus of Communicating Systems*, vol. 92, Berlin, Springer-Verlag, 1980.

[33] Muhlenbein, H., Kramer, O., Limburger, F., Mevenkamp, M. and Streitz, S., "MUPPET: A Programming Environment for Message-Based Multiprocessors," *Parallel Computing*, vol. 8, 1-3 (October 1988), pp. 201-221.

[34] Noe, J. D. and Nutt, G. J., "Macro E-Nets for Representation of Parallel Systems," *IEEE Transactions on Computers*, vol. C-22, 8 (August 1973), pp. 718-727.

[35] Papadopoulos, G. M. and Culler, D. E., "Monsoon: An Explicit Token-Store Architecture," *Proc. 17th Annual Symposium on Computer Architecture, Computer Architecture News*, vol. 18, 2 (June 1990), pp. 82-91.

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

[37] Pratt, V. R., "Modeling Concurrency with Partial Orders," *International Journal of Parallel Programming*, vol. 15, 1 (February 1986), pp. 33-71.

[38] Sabot, G. W., *The Paralation Model: Architecture-Independent Parallel Programming*, Cambridge, MA, MIT Press, 1988.

[39] Sobek, S., Azam, M. and Browne, J. C., "Architectural and Language Independent Parallel Programming: A Feasibility Demonstration," *Proceedings of the 1988 International Conference on Parallel Processing*, vol. II, Software(August 1988), pp. 80-83.

[40] Suhler, P. A., Biswas, J. and Korner, K. M., "TDFL: A Task-Level Data Flow Language," Tech. Rep.-87-44, Austin, TX, University of Texas, CS Dept., November 1987.

Biographical Note

David DiNucci was born in Portland, Oregon on January 13, 1957. He attended Centennial High School in Gresham, Oregon, graduating in 1975. He then attended Portland State University until 1981, receiving a Bachelor of Science Degree in Computer Science. During his stay at PSU, he worked in the computer center as student consultant, computer operator, and programmer. He also worked for one summer at the Harris Corporation in Fort Lauderdale, Florida.

After leaving PSU and spending 4-months in Japan, he took a position with the Portland School District's Research and Evaluation Department, and shortly thereafter married Tamae Sawano. In 1985, after advancing to the position of Data Systems Coordinator, he left the School District to attend Oregon Graduate Institute (then Oregon Graduate Center) full time.

The author is leaving Oregon Graduate Institute to take a summer Post-Doctoral appointment at Lawrence Livermore National Laboratories.